OASIS OpenDocument Essentials

Using OASIS OpenDocument XML

J. David Eisenberg

Cover graphic provided by Peter Harlow

OASIS OpenDocument Essentials:
Using OASIS OpenDocument XML

by J. David Eisenberg

Published by Friends of OpenDocument Inc., P.O. Box 640, Airlie Beach, Qld 4802, Australia, http://friendsofopendocument.org/.

This book was produced using OpenOffice.org 2.0.1. It is printed in the United States of America by Lulu.com (http://www.lulu.com).

The author has a web page for this book, where he lists errata, examples, or any additional information. You can access this page at: http://books.evc-cit.info/index.html . You can download a PDF version of this book at no charge from that website.

ISBN 1-4116-6832-4

Table of Contents

Preface

OASIS OpenDocument Essentials introduces you to the XML that serves as an internal format for office applications. OpenDocument is the native format for OpenOffice.org, an open source, cross-platform office suite, and KOffice, an office suite for KDE (the K desktop environment). It's a format that is truly open and free of any patent and license restrictions.

Who Should Read This Book?

You should read this book if you want to extract data from OpenDocument files, convert your data to OpenDocument format, find out how the format works, or even write your own office applications that support the OpenDocument format.

If you need to know absolutely *everything* about the OpenDocument format, you should download the *Open Document Format for Office Applications (OpenDocument) 1.0* in PDF form from `http://www.oasis-open.org/committees/download.php/12572/OpenDocument-v1.0-os.pdf` or as an OpenOffice.org 1.0 format file from `http://www.oasis-open.org/committees/download.php/12028/office-spec-1.0-cd-3.sxw`. That document was a major source of reference for this book.

Who Should Not Read This Book?

If you simply want to use one of the applications that uses OpenDocument to create documents, you need only download the software and start using it. OpenOffice.org is available at `http://www.openoffice.org/` and KOffice can be found at `http://www.koffice.org/`. There's no need for you to know what's going on behind the scenes unless you wish to satisfy your lively intellectual curiosity.

About the Examples

The examples in this book are written using a variety of tools and languages. I prefer to use open-source tools which work cross-platform, so most of the programming examples will be in Perl or Java. I use the Xalan XSLT processor, which you may find at `http://xml.apache.org`. All the examples in this book have been tested with OpenOffice.org version 1.9.100, Perl 5.8.0, and Xalan-J 2.6.0 on a Linux system using the SuSE 9.2 distribution. This is not to slight any other applications that use OpenDocument (such as KOffice) nor any other operating systems (MacOS X or Windows); it's just that I used the tools at hand.

Conventions Used in This Book

`Constant Width` is used for code examples and fragments.

`Constant width bold` is used to highlight a section of code being discussed in the text.

Constant width italic is used for replaceable elements in code examples.

Names of XML elements will be set in constant width enclosed in angle brackets, as in the `<office:document>` element. Attribute names and values will be in constant width, as in the `fo:font-size` attribute with a value of `0.5cm`.

Sometimes a line of code won't fit on one line. We will split the code onto a second line, but will use an arrow like this ► at the end of the first line to indicate that you should type it all as one line when you create your files.

This book uses *callouts* to denote "points of interest" in code listings. A callout is shown as a white number in a black circle; the corresponding number after the listing gives an explanation. Here's an example:

```
Roses are red,
   Violets are blue.  ❶
Some poems rhyme;
   This one doesn't.  ❷
```

❶ Violets are actually violet. Saying that they are blue is an example of poetic license.

❷ This poem uses the literary device known as a surprise ending.

Acknowledgments

Thanks to Simon St. Laurent, the original editor of this book, who thought it would be a good idea and encouraged me to write it. Thanks also to Erwin Tenhumberg, who suggested that I update the book from the original OpenOffice.org version to the current description of OpenDocument. Thanks also to Adam Moore, who converted the original HTML files to OpenOffice.org format, and to Jean Hollis Weber, who assisted with final layout and proofreading. Edd Dumbill wrote the document which I modified slightly to create Appendix A. Of course, any errors in that appendix have been added by my modifications. Michael Chase provided a platform-independent version of the pack and unpack programs described in the section called "Unpacking and Packing OpenDocument files".

I also want to thank all the people who have taken the time to read and review the HTML version of this book and send their comments. Special thanks to Valden Longhurst, who found a multitude of typographical and grammatical oddities.

—J. David Eisenberg

Chapter 1. The Open Document Format

In this chapter, we will discuss not only the “what” of the OpenDocument format, but also the “why.” Thus, this chapter is as much evangelism as explanation.

The Proprietary World

Before we can talk about OpenDocument, we have to look at the current state of proprietary office suites and applications. In this world, all your documents are stored in a proprietary (often binary) format. As long as you stay within one particular office suite, this is not a problem. You can transfer data from one part of the suite to another; you can transfer text from the word processor to a presentation, or you can grab a set of numbers from the spreadsheet and convert it to a table in your word processing document.

The problems begin when you want to do a transfer that wasn’t intended by the authors of the office suite. Because the internal structure of the data is unknown to you, you can’t write a program that creates a new word processing document consisting of all the headings from a different document. If you need to do something that wasn’t provided by the software vendor, or if you must process the data with an application external to the office suite, you will have to convert that data to some neutral or “universal” format such as Rich Text Format (RTF) or comma-separated values (CSV) for import into the other applications. You have to rely on the kindness of strangers to include these conversions in the first place. Furthermore, some conversions can result in loss of formatting information that was stored with your data.

Note also that your data can become inaccessible when the software vendor moves to a new internal format and stops supporting your current version. (Some people actually suggest that this is not cause for complaint since, by putting your data into the vendor’s proprietary format, the vendor has now become a co-owner of your data. This is, and I mean this in the nicest possible way, a dangerously idiotic idea.)

The OpenDocument Approach

The OpenDocument format has its roots in the XML format used to represent OpenOffice.org files. OpenOffice.org has as its mission "[t]o create, as a community, the leading international office suite that will run on all major platforms and provide access to all functionality and data through open-component based APIs and an XML-based file format." OASIS has taken this format and is advancing its development

The OpenDocument file format is not simply an XML wrapper for a binary format, nor is it a one-to-one correspondence between the XML tags and the internal data structures of a specific piece of application software. Instead, it is an idealized representation of the document's structure. This allows future versions of OpenOffice.org, or any other application that uses OpenDocument, to implement new features or completely alter internal data structures without requiring major changes to the file format. You can see the full details of this design decision at `http://xml.openoffice.org/xml_advocacy.html`

Inside an OpenDocument file

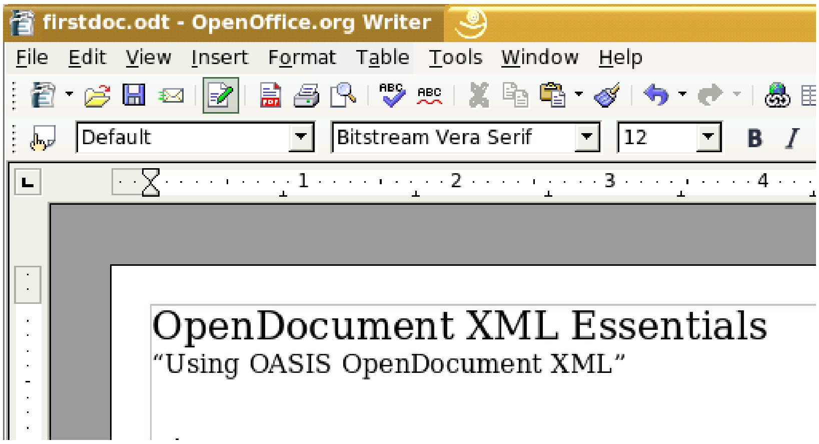
Although the XML file format is human-readable, it is fairly verbose. To save space, OpenDocument files are stored in JAR (Java Archive) format. A JAR file is a compressed ZIP file that has an additional "manifest" file that lists the contents of the archive. Since all JAR files are also ZIP files, you may use any ZIP file tool to unpack an OpenDocument file and read the XML directly.

File or Document?

Because a document in OpenDocument format can consist of several files, saying "an OpenDocument file" is not entirely accurate. However, saying "an OpenDocument document" sounds strange, and "a document in OpenDocument format" is verbose. For purposes of simplicity, when we refer to "an OpenDocument file," we're referring to the whole JAR file, with all its constituent files. When we need to refer to a particular file inside the JAR file, we'll mention it by name.

Figure 1.1, "Text Document" shows a short word processing document, which we have saved with the name `firstdoc.odt`.

Figure 1.1. Text Document


firstdoc.odt - OpenOffice.org Writer
File Edit View Insert Format Table Tools Window Help
Default
Bitstream Vera Serif
12
B I

OpenDocument XML Essentials

"Using OASIS OpenDocument XML"

Example 1.1, "Listing of Unzipped Text Document" shows the results of unzipping this file in Linux; the date, time, and CRC columns have been edited out to save horizontal space. The rows have been rearranged to assist in the explanation.

Example 1.1. Listing of Unzipped Text Document

```
[david@penguin ch01]$ unzip -v firstdoc.odt
Archive:  firstdoc.odt
 Length   Method    Size  Ratio Name
--------  ------  ------- ----- ----
      39  Stored       39   0%  mimetype
    3441  Defl:N      885  74%  content.xml
    6748  Defl:N     1543  77%  styles.xml
    1173  Stored     1173   0%  meta.xml
     642  Defl:N      345  46%  Thumbnails/thumbnail.png
    7176  Defl:N     1307  82%  settings.xml
    1074  Defl:N      308  71%  META-INF/manifest.xml
       0  Stored        0   0%  Configurations2/
       0  Stored        0   0%  Pictures/
--------          -------  ---  -------
   20293             5600  72%  9 files
```

These files are, in order:

`mimetype`
This file has a single line of text which gives the MIME type for the document.The various MIME types are summarized in Table 1.1, "MIME Types and Extensions for OpenDocument Documents".

`content.xml`
The actual content of the document.

`styles.xml`
: This file contains information about the styles used in the content. The content and style information are in different files on purpose; separating content from presentation provides more flexibility.

`meta.xml`
: Meta-information about the content of the document (such things as author, last revision date, etc.) This is different from the `META-INF` directory.

`settings.xml`
: This file contains information that is specific to the application. Some of this information, such as window size/position and printer settings is common to most documents. A text document would have information such as zoom factor, whether headers and footers are visible, etc. A spreadsheet would contain information about whether column headers are visible, whether cells with a value of zero should show the zero or be empty, etc.

`META-INF/manifest.xml`
: This file gives a list of all the other files in the JAR. This is meta-information about the entire JAR file. It is *not* not the same as the manifest file used in the Java language. This file *must* be in the JAR file if you want OpenOffice.org to be able to read it.

`Configurations2`
: I'm not sure what this directory contains!

`Pictures`
: This directory will contain the list of all images contained in the document. Some applications may create this directory in the JAR file even if there aren't any images in the file.

Table 1.1. MIME Types and Extensions for OpenDocument Documents

Document Type	MIME Type	Document Extension
Text document	`application/vnd.oasis.opendocument.text`	`odt`
Text document used as template	`application/vnd.oasis.opendocument.text-template`	`ott`
Graphics document (Drawing)	`application/vnd.oasis.opendocument.graphics`	`odg`
Drawing document used as template	`application/vnd.oasis.opendocument.graphics-template`	`otg`
Presentation document	`application/vnd.oasis.opendocument.presentation`	`odp`
Presentation document used as template	`application/vnd.oasis.opendocument.presentation-template`	`otp`
Spreadsheet document	`application/vnd.oasis.opendocument.spreadsheet`	`ods`
Spreadsheet document used as template	`application/vnd.oasis.opendocument.spreadsheet-template`	`ots`
Chart document	`application/vnd.oasis.opendocument.chart`	`odc`
Chart document used as template	`application/vnd.oasis.opendocument.chart-template`	`otc`
Image document	`application/vnd.oasis.opendocument.image`	`odi`
Image document used as template	`application/vnd.oasis.opendocument.image-template`	`oti`
Formula document	`application/vnd.oasis.opendocument.formula`	`odf`
Formula document used as template	`application/vnd.oasis.opendocument.formula-template`	`otf`
Global Text document	`application/vnd.oasis.opendocument.text-master`	`odm`
Text document used as template for HTML documents	`application/vnd.oasis.opendocument.text-web`	`oth`

We will discuss the `meta.xml`, `settings.xml`, and `style.xml` files in greater detail in the next chapter, and the remainder of the book will cover the various flavors of the `content.xml` file.

The manifest.xml File

First, let's look at the contents of `manifest.xml`, most of which is self-explanatory.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE manifest:manifest
        PUBLIC "-//OpenOffice.org//DTD Manifest 1.0//EN"
"Manifest.dtd">
<manifest:manifest
        xmlns:manifest="urn:oasis:names:tc:opendocument:xmlns:►
manifest:1.0">
 <manifest:file-entry
        manifest:media-type="application/vnd.oasis.opendocument.text"
        manifest:full-path="/"/>
 <manifest:file-entry
        manifest:media-type="application/vnd.sun.xml.ui.configuration"
        manifest:full-path="Configurations2/"/>
 <manifest:file-entry
        manifest:media-type="" manifest:full-path="Pictures/"/>
 <manifest:file-entry
        manifest:media-type="text/xml"
        manifest:full-path="content.xml"/>
 <manifest:file-entry
        manifest:media-type="text/xml"
        manifest:full-path="styles.xml"/>
 <manifest:file-entry
        manifest:media-type="text/xml" manifest:full-path="meta.xml"/>
 <manifest:file-entry
        manifest:media-type=""
        manifest:full-path="Thumbnails/thumbnail.png"/>
 <manifest:file-entry
        manifest:media-type="" manifest:full-path="Thumbnails/"/>
 <manifest:file-entry
        manifest:media-type="text/xml"
        manifest:full-path="settings.xml"/>
</manifest:manifest>
```

The `manifest:media-type` for the root directory tells what kind of file this is. Its content is the same as the content of the `mimetype` file, as shown in Table 1.1, "MIME Types and Extensions for OpenDocument Documents", adapted from the OpenDocument specification.

There is an entry for a `Pictures` directory, even though there are no images in the file. If there were an image, the unzipped file would contain a `Pictures` directory, and the relevant portion of the manifest would now look like this:

```
    <manifest:file-entry manifest:media-type="image/png"
        manifest:full-►
path="Pictures/10000200000002000000020DF8717E9.png" />
    <manifest:file-entry manifest:media-type=""
        manifest:full-path="Pictures/" />
```

If you are using OpenOffice.org and have included OpenOffice.org BASIC scripts, your packed file will include a `Basic` directory, and the manifest will describe it and its contents.

If you are building your own document with embedded objects (charts, pictures, etc.) you must keep track of them in the manifest file, or OpenOffice.org will not be able to find them.

Namespaces

The `manifest.xml` used the `manifest` namespace for all of its element and attribute names. OpenDocument uses a large number of namespace declarations in the root element of the `content.xml`, `styles.xml`, and `settings.xml` files. Table 1.2, “Namespaces for OpenDocument”, which is adapted from the OpenDocument specification, shows the most important of these.

Table 1.2. Namespaces for OpenDocument

Namespace Prefix	Describes	Namespace URI
`office`	Common information not contained in another, more specific namespace.	`urn:oasis:names:tc:opendocument:xmlns:office:1.0`
`meta`	Meta information.	`urn:oasis:names:tc:opendocument:xmlns:meta:1.0`
`config`	Application-specific settings.	`urn:oasis:names:tc:opendocument:xmlns:config:1.0`
`text`	Text documents and text parts of other document types (e.g., a spreadsheet cell).	`urn:oasis:names:tc:opendocument:xmlns:text:1.0`
`table`	Content of spreadsheets or tables in a text document.	`urn:oasis:names:tc:opendocument:xmlns:table:1.0`
`drawing`	Graphic content.	`urn:oasis:names:tc:opendocument:xmlns:drawing:1.0`
`presentation`	Presentation content.	`urn:oasis:names:tc:opendocument:xmlns:presentation:1.0`
`dr3d`	3D graphic content.	`urn:oasis:names:tc:opendocument:xmlns:dr3d:1.0`
`anim`	Animation content.	`urn:oasis:names:tc:opendocument:xmlns:animation:1.0`
`chart`	Chart content.	`urn:oasis:names:tc:opendocument:xmlns:chart:1.0`
`form`	Forms and controls.	`urn:oasis:names:tc:opendocument:xmlns:form:1.0`

Namespace Prefix	Describes	Namespace URI
`script`	Scripts or events.	`urn:oasis:names:tc:opendocument:xmlns:script:1.0`
`style`	Style and inheritance model used by OpenDocument; also common formatting attributes.	`urn:oasis:names:tc:opendocument:xmlns:style:1.0`
`number`	Data style information.	`urn:oasis:names:tc:opendocument:xmlns:data style:1.0`
`manifest`	The package manifest.	`urn:oasis:names:tc:opendocument:xmlns:manifest:1.0`
`fo`	Attributes defined in XSL:FO.	`urn:oasis:names:tc:opendocument:xmlns:xsl-fo-compatible:1.0`
`svg`	Elements or attributes defined in SVG.	`urn:oasis:names:tc:opendocument:xmlns:svg-compatible:1.0`
`smil`	Attributes defined in SMIL20.	`urn:oasis:names:tc:opendocument:xmlns:smil-compatible:1.0`
`dc`	The Dublin Core Namespace.	`http://purl.org/dc/elements/1.1/`
`xlink`	The XLink namespace.	`http://www.w3.org/1999/xlink`
`math`	MathML Namespace.	`http://www.w3.org/1998/Math/MathML`
`xforms`	The XForms namespace.	`http://www.w3.org/2002/xforms`
`xforms`	The WWW Document Object Model namespace.	`http://www.w3.org/2001/`► `xml-events`
`ooo`	The OpenOffice.org namespace.	`http://openoffice.org/2004/`► `office`
`ooow`	The OpenOffice.org writer namespace.	`http://openoffice.org/2004/`► `writer`
`ooo`	The OpenOffice.org spreadsheet (calc) namespace.	`http://openoffice.org/2004/calc`

Whenever possible, OpenDocument uses existing standards for namespaces. The `text` namespace adds elements and attributes that describe the aspects of word processing that the `fo` namespace lacks; similarly `draw` and `dr3d` add functionality that is not already found in `svg`.

Unpacking and Packing OpenDocument files

If you unzip an OpenDocument file, it will unzip into the current directory. If you unpack a second document, your unzip program will either overwrite the old files or prompt you at each file. This is inconvenient, so we have written a Perl program, shown in Example 1.2, “Program to Unpack an OpenDocument File”, which will unpack an OpenDocument file whose name has the form `filename.extension`. It will unzip the files into a directory named `filename_extension`. You will find this program as file `odunpack.pl` in directory `ch01` in the downloadable example files.

Example 1.2. Program to Unpack an OpenDocument File

```
#!/usr/bin/perl

#
#   Unpack an OpenDocument file to a directory.

#
#   Archive::Zip is used to unzip files.
#   File::Path is used to create and remove directories.
#
use Archive::Zip;
use File::Path;
use strict;

my $file_name;
my $dir_name;
my $suffix;
my $zip;
my $member_name;
my @member_list;

if (scalar @ARGV != 1)
{
    print "Usage: $0 filename\n";
    exit;
}

$file_name = $ARGV[0];

#
#   Only allow filenames that have valid OpenDocument extensions
#
if ($file_name =~ m/\.(o[dt][tgpscif]|odm|oth)/)
{
    $suffix = $1;

    #
    #   Create directory name based on filename
    #
    ($dir_name = $file_name) =~ s/\.$suffix//;
    $dir_name .= "_$suffix";

    #
    #       Forcibly remove old directory, re-create it,
    #       and unzip the OpenOffice.org file into that directory
    #
    rmtree($dir_name, 0, 0);
    mkpath($dir_name, 0, 0755);

    $zip = Archive::Zip->new( $file_name );
    @member_list = $zip->memberNames( );

    foreach $member_name (@member_list)
    {
        $zip->extractMember( $member_name,
        "$dir_name/$member_name" );
    }

    print "$file_name unpacked.\n";
}
else
{
    print "This does not appear to be an OpenDocument file.\n";
    print "Legal suffixes are .odt, .ott, .odg, .otg, .odp, .otp,\n";
    print ".ods, .ots, .odc, .otc, .odi, .oti, .odf, .otf, .odm,►
and .oth\n";
}
```

When you look at the unpacked files in a text editor, you will notice that most of them consist of only two lines: a `<!DOCTYPE>` declaration followed by a single line containing the rest of the document. Ordinarily this is no problem, as the documents are meant to be read by a program rather than a human. In order to analyze the XML files for this book, we had to put the files in a more readable format. In OpenOffice.org, this was easily accomplished by turning off the “Size optimization for XML format (no pretty printing)” checkbox in the Options—Load/Save—General dialog box. All the files we created from that point onward were nicely formatted. If you are receiving files from someone else, and you do not wish to go to the trouble of opening and re-saving each of them, you may use XSLT to do the indenting, as explained in the section called “Using XSLT to Indent OpenDocument Files”.

If you need to pack (or repack) files to produce a single OpenDocument file, Example 1.3, “Program to Pack Files to Create an OpenDocument File” does exactly that. It takes the files in a directory of the form *filename_extension* and creates a document named `filename.extension` (or any other name you wish to give as a second argument on the command line). You will find this program as file `odpack.pl` in directory `ch01` in the downloadable example files.

Example 1.3. Program to Pack Files to Create an OpenDocument File

```
#!/usr/bin/perl

#
#   Repack a directory to an OpenDocument file
#
#   Directory xyz_odt will be packed into xyz.odt, etc.
#
#
```

```
use Archive::Zip;        # to zip files
use Cwd;                 # to get current working directory

use strict;

my $dir_name;            # directory name to zip
my $file_name = "";      # destination file name
my $suffix;                      # file extension
my $current_dir;         # current directory
my $zip;                         # a zip file object

if (scalar @ARGV < 1 || scalar @ARGV > 2)
{
    print "Usage: $0 directoryname [newfilename]\n";
    exit;
}

$dir_name = $ARGV[0];

#
#   If no new filename is given, create a filename
#   based on directory name
#

if ($ARGV[1])
{
    $file_name = $ARGV[1];
}
else
{
    if ($dir_name =~ m/_(o[dt][tgpscif]|odm|oth)/)
    {
        $suffix = $1;
        ($file_name = $dir_name) =~ s/(_$suffix)//;
        $file_name .= ".$suffix";
    }
    else
    {
        print "This does not appear to be an unpacked OpenDocument►
file.\n";
        print "Legal suffixes are _odt, _ott, _odg, _otg, _odp, _otp,►
_ods,\n";
        print "_ots, _odc, _otc, _odi, _oti, _odf, _otf, _odm, and►
_oth\n";
        $file_name = "";
    }
}

if ($file_name ne "")
{
    $zip = Archive::Zip->new();

    $current_dir = cwd();

    if (chdir($dir_name))
    {
        $zip->addTree( '.' );
        $zip->writeToFileNamed( "../$file_name" );
        print "$dir_name packed to $file_name.\n";
        chdir($current_dir);
    }
    else
    {
        print "Could not change directory to $dir_name\n";
    }
}
```

The Virtues of Cheating

As you begin to work with OpenDocument files, you may want to write a program that constructs a document with some feature that isn’t explained in this book—this is, after all, an “essentials” book. Just start OpenOffice.org or KOffice, create a document that has the feature you want, unpack the file, and look for the XML that implements it. To get a better understanding of how things works, change the XML, repack the document, and reload it. Once you know how a feature works, don’t hesitate to copy and paste the XML from the OpenDocument file into your program. In other words, cheat. It worked for me when I was writing this book, and it can work for you too!

Chapter 2. The meta.xml, styles.xml, settings.xml, and content.xml Files

Though `content.xml` is king, monarchs rule better when surrounded by able assistants. In an OpenDocument JAR file, these assistants are the `meta.xml`, `style.xml`, and `settings.xml` files. In this chapter, we will examine the assistant files, and then describe the general structure of the `content.xml` file.

The only files that are actually necessary are `content.xml` and the `META-INF/manifest.xml` file. If you create a file that contains word processor elements and zip it up and a manifest that points to that file, OpenOffice.org will be able to open it successfully. The result will be a plain text-only document with no styles. You won't have any of the meta-information about who created the file or when it was last edited, and the printer settings, view area, and zoom factor will be set to the OpenOffice.org defaults.

The settings.xml File

The `settings.xml` file contains information intended for use exclusively by the application that created the file. From the viewpoint of an external application, there's very little of use in this file, so we'll just take a brief look at it before bidding it a fond farewell.

The root element, `<office:document-settings>` contains a `<office:settings>` element, which in turn contains one or more `<config:config-item-set>` entries. Each of these contains one or more items, named item maps,indexed item maps, or other `<config:config-item-set>`s.

Configuration Items

The `<config:config-item>` element has a `config:name` attribute that describes the item and a `config:type` attribute which can be one of `boolean`, `short`, `int`, `long`, `double`, `string`, `datetime`, or `base64Binary`. The content of the element gives the value of that item. Example 2.1, "Example of Configuration Items" shows some representative configuration items from a word processing document:

Example 2.1. Example of Configuration Items

```
<config:config-item config:name="PrinterName"
   config:type="string">Generic Printer</config:config-item>

<config:config-item config:name="ViewLeft"
   config:type="int">2043</config:config-item>

<config:config-item config:name="ShowRedlineChanges"
   config:type="boolean">true</config:config-item>
```

Named Item Maps

The `<config:config-item-map-named>` element contains one or more `<config:config-item-map-entry>` sub-elements. Each of these map entries may contain one or more items, item sets, named item maps, or indexed item maps (yes, this is a very recursive data structure). Entries in a named item map are accessed by their unique `name` attribute. Spreadsheets use a named item map to store information about of each of the sheets in the document.

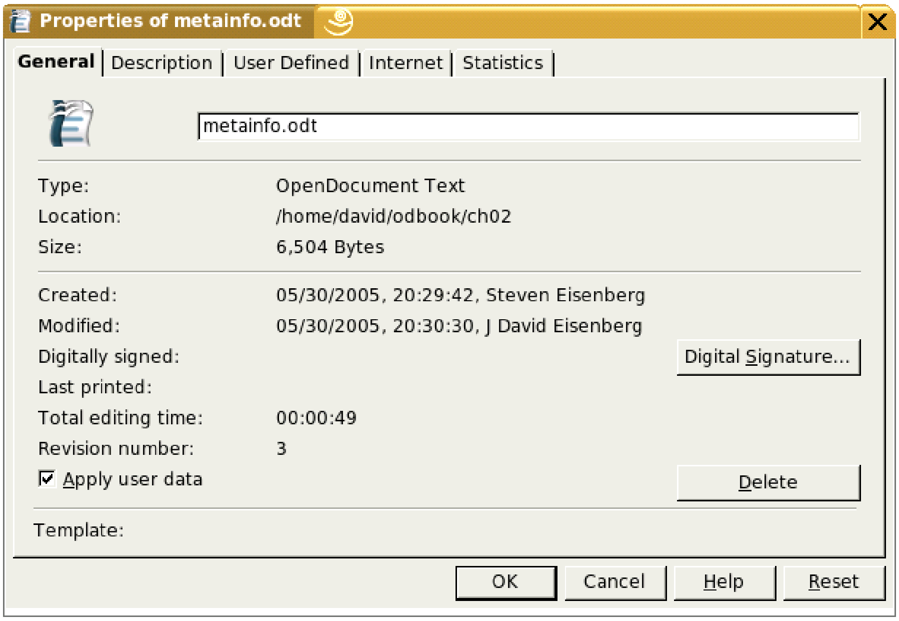
Indexed Item Maps

A `<config:config-item-map-indexed>` element also contains one or more `<config:config-item-map-entry>` elements. Each of these map entries may contain one or more items, item sets, named item maps, or indexed item maps. The order of the individual map entries is important; entries are accessed by their position, not by their unique `name` attribute.

The meta.xml File

The `meta.xml` file contains information about the document itself. We'll look at the elements found in this file in decreasing order of importance; at the end of this section, we will list them in the order in which they appear in a document. Most of these elements are reflected in the tabs on OpenOffice.org's File/Properties dialog, which are show in Figure 2.1, "General Document Properties", Figure 2.2, "Document Description", Figure 2.3, "User-defined Information", and Figure 2.4, "Document Statistics".

Figure 2.1. General Document Properties



Properties of metainfo.odt

General | Description | User Defined | Internet | Statistics

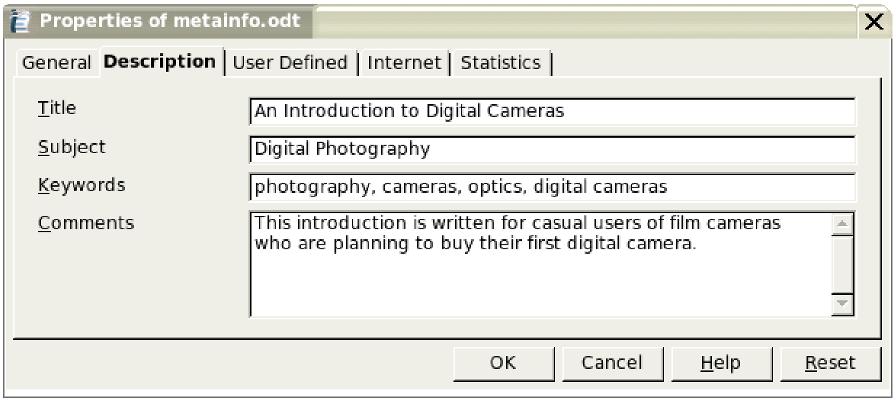
metainfo.odt

Type:	OpenDocument Text
Location:	/home/david/odbook/ch02
Size:	6,504 Bytes
Created:	05/30/2005, 20:29:42, Steven Eisenberg
Modified:	05/30/2005, 20:30:30, J David Eisenberg
Digitally signed:	
Last printed:	
Total editing time:	00:00:49
Revision number:	3

☑ Apply user data

Template:

Figure 2.2. Document Description

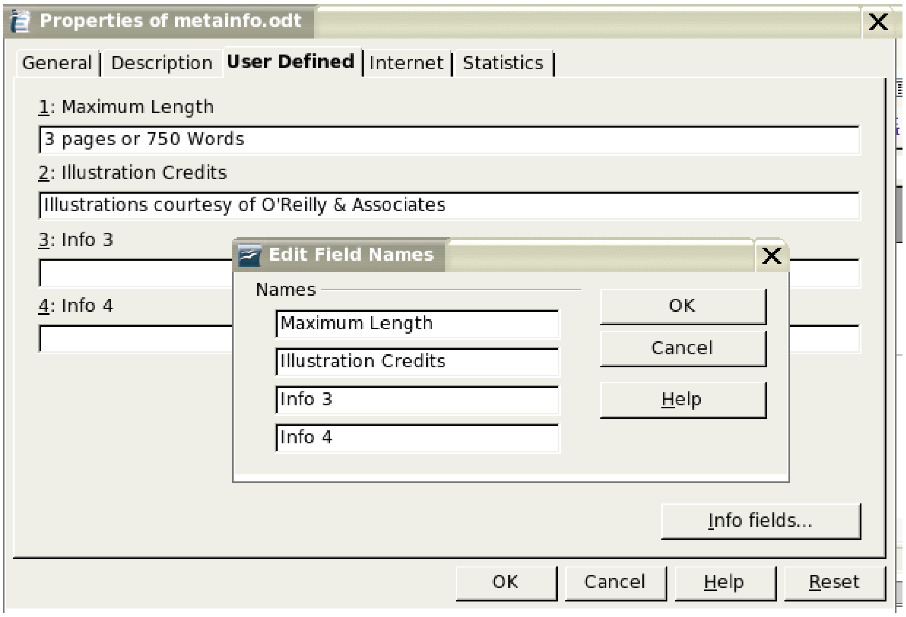


Properties of metainfo.odt

General | Description | User Defined | Internet | Statistics

Title	An Introduction to Digital Cameras
Subject	Digital Photography
Keywords	photography, cameras, optics, digital cameras
Comments	This introduction is written for casual users of film cameras who are planning to buy their first digital camera.

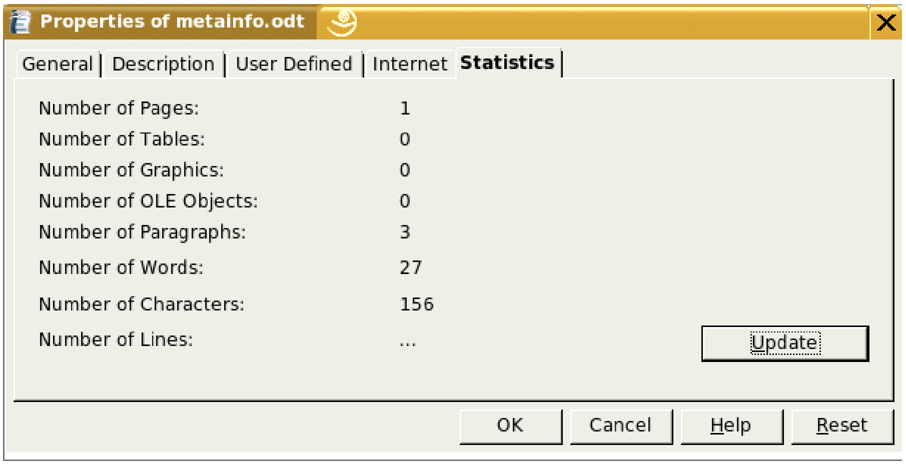
Figure 2.3. User-defined Information



1: Maximum Length
3 pages or 750 Words
2: Illustration Credits
Illustrations courtesy of O'Reilly & Associates
3: Info 3
4: Info 4

Maximum Length
Illustration Credits
Info 3
Info 4

Figure 2.4. Document Statistics



Number of Pages:	1
Number of Tables:	0
Number of Graphics:	0
Number of OLE Objects:	0
Number of Paragraphs:	3
Number of Words:	27
Number of Characters:	156
Number of Lines:	...

The Dublin Core Elements

All elements borrowed from the Dublin Core namespace contain text and have no attributes. Table 2.1, "Dublin Core Elements in meta.xml" summarizes them.

Table 2.1. Dublin Core Elements in meta.xml

Element	Description	Sample from XML file
`<dc:title>`	The document title; this appears in the title bar.	`<dc:title>An Introduction to Digital Cameras</dc:title>`
`<dc:subject>`	The Dublin Core recommends that this element contain keywords or key phrases to describe the topic of the document; OpenOffice.org keeps keywords in a separate set of elements.	`<dc:subject>Digital Photography</dc:subject>`
`<dc:description>`	This element's content is shown in the Comments field in the dialog box.	`<dc:description>This introduction… </dc:description>`
`<dc:creator>`	This element's content is shown in the Modified field in Figure 2.1, "General Document Properties"; it names the last person to edit the file. This may appear odd, but the Dublin Core says that the creator is simply an "entity primarily responsible for making the content of the resource." That is not necessarily the original creator, whose name is stored in a different element.	`<dc:creator>J David Eisenberg</dc:creator>`
`<dc:date>`	This element's content is also shown in the Modified field in Figure 2.1, "General Document Properties". It is stored in a form compatible with ISO-8601. The time is shown in local time. See the section called "Time and Duration Formats" for details about times and dates.	`<dc:date>2005-05-30T20:30:30</dc:date>`
`<dc:language>`	The document's language, written as a two or three-letter main language code followed by a two-letter sublanguage code. This field is not shown in the properties dialog, but is found in OpenOffice.org's Tools/Options/ Language Settings dialog.	`<dc:language>en-US</dc:language>`

Elements from the meta Namespace

The remaining elements in the `meta.xml` file come from OpenDocument's `meta` namespace. Table 2.2, "OpenDocument Elements in meta.xml" describes these elements in the order in which they appear in the file.

Table 2.2. OpenDocument Elements in meta.xml

Element	Description	Sample from XML file
`<meta:generator>`	The program that created this document. According to the specifcation, you should not "fake" being OpenOffice.org if you are creating the document using a different program; you should use a unique identifier.	`<meta:generator>OpenOffice.org/1.9.100$Linux OpenOffice.org_project/680m100$Build-8909</meta:generator>`
`<meta:initial-creator>`	The user who created the document. This is shown in the "Created:" area in Figure 2.1, "General Document Properties".	`<meta:initial-creator>Steven Eisenberg</meta:initial-creator>`
`<meta:creation-date>`	The date and time when the document was created. This is shown in the "Created:" area in Figure 2.1, "General Document Properties". It is in the same format as described in the section called "Time and Duration Formats".	`<meta:creation-date>2005-05-30T20:29:42</meta:creation-date>`
`<meta:keyword>`	A document may contain one or more `<meta:keyword>` elements. These elements reflect the entries in the "Keywords:" area in Figure 2.2, "Document Description".	`<meta:keyword>photography</meta:keyword>` `<meta:keyword>cameras</meta:keyword>` `<meta:keyword>optics</meta:keyword>` `<meta:keyword>digital cameras</meta:keyword>` `</meta:keywords>`
`<meta:editing-cycles>`	This element tells how many times the file has been edited; this is the " Revision Number:" in in Figure 2.1, "General Document Properties".	`<meta:editing-cycles>5</meta:editing-cycles>`
`<meta:editing-duration>`	This element tells the total amount of time that has been spent editing the document in all editing sessions; this is the "Editing time:" in Figure 2.1, "General Document Properties", and is represented as described in the section called "Time and Duration Formats".	`<meta:editing-duration>PT1H28M55S</meta:editing-duration>`

Element	Description	Sample from XML file
`<meta:user-defined>`	OpenOffice.org allows you to define your own information, as shown in Figure 2.3, "User-defined Information". This element has a `meta:name` attribute, giving the "title" of this information, and the content of the element is the information itself.	`<meta:user-defined meta:name="Maximum Length">3 pages or 750 words</meta:user-defined>`
`<meta:document-statistic>`	This is the information shown on the statistics tab of the properties dialog (see Figure 2.4, "Document Statistics"). This element has attributes whose names are largely self-explanatory, and are listed in Table 2.3, "Attributes of the <meta:document-statistic> Element".	`<meta:document-statistic meta:paragraph-count="4"…/>`

Table 2.3. Attributes of the <meta:document-statistic> Element

Attribute	Description
`meta:page-count`	Number of pages in a word processing document. This must be greater than zero. This attribute is not used in spreadsheets. The "number of pages" shown in the statistics dialog for a spreadsheet is a calculated value that tells how many sheets have filled cells on them, and this can be zero for a totally empty spreadsheet.
`meta:paragraph-count`	Number of paragraphs in a word processing document.
`meta:word-count`	Number of words in a word processing document.
`meta:character-count`	Number of characters in a word processing document.
`meta:image-count`	Number of images in a word processing document.
`meta:table-count`	Number of tables in a word processing document, or number of sheets in a spreadsheet document.
`meta:cell-count`	Number of non-empty cells in a spreadsheet document.
`meta:object-count`	Number of objects in a document. This is shown as "Number of OLE objects" in the dialog box of Figure 2.4, "Document Statistics". This attribute is used in drawing and presentation documents, but it does not bear any simple relationship to the number of items you see on the screen.
`meta:ole-object-count`	Apparently unused in OpenOffice.org2.0.
`meta:row-count`	Apparently unused in OpenOffice.org2.0.
`meta:draw-count`	Apparently unused in OpenOffice.org2.0.

Time and Duration Formats

The dates, times, and durations used in the metadata are patterned after the format described in the ISO 8601 standard. A date is written as a four-digit year, two-digit month, and two-digit day separated by hyphens. The capital letter `T` separates the date from the time, which is written in the form *hh*`:`*mm*`:`*ss*.

Warning
OpenOffice.org does not implement the full ISO 8601 standard. For example, you may not use a truncated form such as `--06-20` for a date, nor may you add a time zone offset after the time.

When you insert a date or time field into a text document, the seconds field is followed by a comma and decimal fraction of a second. Thus, `2005-06-01T09:54:26,50` represents 9:54 and 26.5 seconds on the 1st of June, 2005.

Time durations, such as those in the `<meta:editing-duration>` element, describe a length of days, hours, minutes, and seconds, written in the form `P`*d*`DT`*h*`H`*m*`M`*s*`S`. If the editing time is less than one day, the *d*`D` is omitted. Thus, `PT12M34S` describes a duration of twelve minutes and thirty-four seconds. A duration may not specify a number of years or months as described in the ISO 8601 standard.

Case Study: Extracting Meta-Information

Now that we know what the format of the meta file is, let's construct a Perl program to extract that information. Again, rather than reinvent the wheel, we will use two existing modules from the Comprehensive Perl Archive Network, CPAN (`http://www.cpan.org/`). The first of these, Archive::Zip::MemberRead, will let us read the `meta.xml` file directly from a compressed OpenDocument document. We will use the XML::Simple module to do the main work of the extraction program.

Archive::Zip::MemberRead

It would be inefficient to unpack the document into a temporary directory, read the `meta.xml` file, and then remove the temporary directory. Archive::Zip::MemberRead avoids this problem by letting you read members of a ZIP archive without having to unpack them. Example 2.2, "Program member_read.pl" shows the program that uses this module. The program takes two arguments: the OpenDocument file name and the member of the packed document that you want to read. The program parcels out the data in 32 kilobyte chunks rather than reading it in as one huge string. It also sends its output to standard output so that it can be piped to other processes. [This is program `member_read.pl` in directory `ch02` in the downloadable example files.]

Example 2.2. Program member_read.pl

```
#!/usr/bin/perl

use Archive::Zip;
use Archive::Zip::MemberRead;
use Carp;
use strict 'vars';

my $zip;        # the zip file
my $fh;         # filehandle to the member being read
my $buffer;     # 32 kilobyte buffer

#
#   Extract a single XML file from an OpenOffice.org file
#   Output goes to standard output
#
if (scalar @ARGV != 2)
{
    croak("Usage: $0 document xmlfilename");
}

$zip = new Archive::Zip($ARGV[0]);
if (!$zip)
{
    croak("$ARGV[0] cannot be opened as a .ZIP file");
}

$fh  = new Archive::Zip::MemberRead($zip, $ARGV[1]);
if (!$fh)
{
    croak("$ARGV[0] does not contain a file named $ARGV[1]");
}

while ($fh->read($buffer, 32*1024))
{
    print $buffer;
}
```

XML::Simple

This module does all the heavy work of parsing an XML file. You just give it the name of a file to parse, and it returns a reference to a hash which contains nested hashes. For example, given this document:

```
<document>
    <h1>Information</h1>
    <para align="left" style="color:red;">More info</para>
</document>
```

XML::Simple returns a data structure as if we had written this Perl statement:

```
$result = {
  'h1' => 'Information',
  'para' => {
              'align' => 'left',
              'content' => 'More info',
              'style' => 'color:red;'
            }
};
```

The Meta Extraction Program

The program that actually does the extraction, Example 2.3, “Program show_meta.pl”, takes one argument: the OpenDocument filename. The program receives its input from the piped output of `member_read.pl`.

After the file is parsed, the program prints the data. Information in the `<meta:document-statistic>` is selected depending upon the type of document being parsed. The program also uses the Text::Wrap module to format the description, which may be several lines long. [This is program `show_meta.pl` in directory `ch02` in the downloadable example files.]

Example 2.3. Program show_meta.pl

```
#!/usr/bin/perl

#
#   Show meta-information from an OpenDocument meta.xml file.
#
use XML::Simple;
use IO::File;
use Text::Wrap;
use Carp;
use strict;

my $suffix;     # file suffix

#
#   Check for one argument: the name of the OpenOffice.org document
#
if (scalar @ARGV != 1)
{
    croak("Usage: $0 document");
}

#
#   Get file suffix for later reference
#
($suffix) = $ARGV[0] =~ m/\.(\w\w\w)$/;

#
#   Parse and collect information into the $meta hash reference
#
$ARGV[0] =~ s/[;|'"]//g;  #eliminate dangerous shell metacharacters
my $fh = IO::File->new("perl member_read.pl $ARGV[0] meta.xml |");
my $xml= XMLin( $fh, forcearray => [ 'meta:keyword'] );
my $meta= $xml->{'office:meta'};

#
#   Output phase
#
print "Title:       $meta->{'dc:title'}\n"
    if ($meta->{'dc:title'});
print "Subject:     $meta->{'dc:subject'}\n"
    if ($meta->{'dc:subject'});

if ($meta->{'dc:description'})
{
    print "Description:\n";
    $Text::Wrap::columns = 60;
    print wrap("\t", "\t", $meta->{'dc:description'}), "\n";
}

print "Created:     ";
print format_date($meta->{'meta:creation-date'});
print " by $meta->{'meta:initial-creator'}"
    if ($meta->{'meta:initial-creator'});
print "\n";

print "Last edit:   ";
print format_date($meta->{"dc:date"});
print " by $meta->{'dc:creator'}"
    if ($meta->{'dc:creator'});
print "\n";

# Display keywords (which all appear to be in a single element)
#
print "Keywords:    ", join( ' - ',
  @{$meta->{'meta:keywords'}->{'meta:keyword'}}), "\n"
    if( $meta->{'meta:keywords'});

#
#   Take attributes from the meta:document-statistic element
#   (if any) and put them into the $statistics hash reference
#
my $statistics= $meta->{'meta:document-statistic'};
if ($suffix eq "sxw")
{
        print "Pages:       $statistics->{'meta:page-count'}\n";
        print "Words:       $statistics->{'meta:word-count'}\n";
        print "Tables:      $statistics->{'meta:table-count'}\n";
        print "Images:      $statistics->{'meta:image-count'}\n";
}
elsif ($suffix eq "sxc")
{
        print "Sheets:      $statistics->{'meta:table-count'}\n";
        print "Cells:       $statistics->{'meta:cell-count'}\n"
                if ($statistics->{'meta:cell-count'});
}

#
#   A convenience subroutine to make dates look
#   prettier than ISO-8601 format.
#
sub format_date
{
    my $date = shift;
    my ($year, $month, $day, $hr, $min, $sec);
    my @monthlist = qw (Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov
Dec);

    ($year, $month, $day, $hr, $min, $sec) =
        $date =~ m/(\d{4})-(\d{2})-(\d{2})T(\d{2}):(\d{2}):(\d{2})/;
    return "$hr:$min on $day $monthlist[$month-1] $year";
}
```

These two lines from the preceding program are where all the parsing takes place:

```
my $fh = IO::File->new("perl member_read.pl $ARGV[0] meta.xml |");
my $xml= XMLin( $fh, forcearray => [ 'meta:keyword'] );
```

In the first line, we used `IO::File->new`, because our version of Perl wouldn't read from a file handle opened with the standard Perl `open()`. In the second line, the `forcearray` parameter will force the content of the `<meta:keyword>` element to be an array type, even if there is only one element. This avoids scalar versus array problems.

While XML::Simple is the easiest way to accomplish this task, it is not the most flexible way to parse XML. For more general XML parsing, you probably want to use the XML::SAX module. the section called “Showing Meta-information Using SAX” shows this same program written with the XML::SAX module.

The styles.xml File

The `styles.xml` file contains information about the styles that are used in the document. Some of this information is also duplicated in the `content.xml` document.

File `styles.xml` begins with a `<office:document-styles>` element, which contains font declarations (`<office:font-decls>`), default and named styles (`<office:styles>`), "automatic," or unnamed styles (`<office:automatic-styles>`), and master styles (`<office:master-styles>`). All of these elements are optional.

Font Declarations

The `<office:font-face-decls>` element contains zero or more `<style:font-face>` elements. `<style:font-face>` is an empty element, some of whose attributes are described in Table 2.4, “Attributes of the <style:font-face> Element”.

Table 2.4. Attributes of the <style:font-face> Element

Attribute	Description
`style:name`	The name of the font (required).
`svg:font-family`	The font family (optional). It is not necessarily the same as the font name. For example, a font named Courier Bold Oblique belongs to the Courier family, and its `svg:font-family` attribute would be `Courier`. If a font family name has blanks in it, such as `Zapf Chancery`, OpenOffice.org encloses the value in single quotes.
`style:font-family-generic`	The generic class to which this font belongs. Valid values for this optional attribute are `roman` (serif), `swiss` (sans-serif), `modern`, `decorative`, `script`, and `system`.

Attribute	Description
`style:font-pitch`	This optional attribute tells whether the font is `fixed` (fixed-width, as is the Courier font) or `variable` (proportional-width).
`style:font-charset`	The encoding for this font; this attribute is optional.

There is also a large number of attributes borrowed from SVG, such as `svg:font-stretch`, `svg:units-per-em`, `svg:ascent`, but current applications that create OpenDocument documents don't appear to use them.

Office Default and Named Styles

The `<office:styles>` element is a container for (among other things) default styles and named styles. In OpenOffice.org, these are set with the Stylist tool. A spreadsheet's `<office:styles>` element will also contain information about style for numbers, currency, percentage values, dates, times, and boolean data. A drawing will have information about default gradients, hatch patterns, fill images, markers, and dash patterns for drawing lines.

The most important elements that you will find within `<office:styles>` are `<style:default-style>` and `<style:style>`. Both elements contain a `style:family` attribute which tells what "level" the style applies to. The possible values of this required attribute are `text` (character level), `paragraph`, `section`, `table`, `table-column`, `table-row`, `table-cell`, `table-page`, `chart`, `graphics`, `default`, `drawing-page`, `presentation`, `control`, and `ruby`[1]

Both `<style:default-style>` and `<style:style>` have a `style:name` attribute. Styles built in to OpenOffice.org's stylist, or ones that you create there, will have names like `Heading_20_1` or `Custom_20_Citation`. Non-alphanumeric characters in names are converted to hexadecimal; thus blanks are converted to `_20_`. A style named `!wow?#@$` would be stored as `_21_wow_3f__23__40__24_`. Automatic styles will have names consisting of a one- or two-letter abbreviation followed by a number; a style name such as `T1` is the first automatic style for `style:family="text"`; `P3` would be the third style for paragraphs, `ta2` would be the second style for a table, `ro4` would be the fourth style for a table row, etc.

[1] Ruby refers to "furigana," which are small Japanese alphabetic characters placed near the Japanese ideograms to aid readers in determining their correct meaning.

Names and Display Names

Internal names are stored in the `style:name` attribute, with non-alphanumeric characters translated to their hexadecimal equivalents. If there are any non-numeric characters, OpenDocument also provides a `style:display-name` attribute that gives the unencoded version of the name, suitable for display to a user in an application. Thus, the encoded `style:name="_21_wow_3f__23__40__24_"` has the display form `style:display-name="!wow?#@$"`.

You will see this pairing of name and display-name in attributes in graphics as `draw:name` and `draw:display-name`.

The other attribute of interest is the optional `parent-style-name`, which you will find in styles that have been derived from other styles. In a text document, OpenOffice.org will often create a temporary style whose parent is the style found in the `styles.xml` file.

Within each `<style:style>` or `<style:default-style>`, you will find the `<style:`*`family`*`-properties>` element, which describes the style in minute detail via an immense number of attributes. The *family* is related to the `style:family` attribute; if a style has `style:family="table"`, then it will contain a `<style:table-properties>` element; `style:family="paragraph"`, will contain a `<style:paragraph-properties>` element, and so forth.

A full discussion of styles is beyond the scope of this book, so we will simply give you an idea of the range of style specifications, and take up specific details of styles when they are relevant in other chapters. Example 2.4, “Style Defintion in a Word Processing Document”, Example 2.5, “Style Defintion in a Spreadsheet Document”, and Example 2.6, “Style Defintion in a Drawing Document” are excerpts from the `styles.xml` files in a word processing, spreadsheet, and drawing document

Example 2.4. Style Defintion in a Word Processing Document

```
<style:style style:name="Text_20_body"
    style:display-name="Text body" style:family="paragraph"
    style:parent-style-name="Standard" style:class="text">
    <style:paragraph-properties
        fo:margin-top="0in" fo:margin-bottom="0.0835in"/>
</style:style>
```

Example 2.5. Style Defintion in a Spreadsheet Document

```
<number:currency-style style:name="N104">
    <style:text-properties fo:color="#ff0000"/>
    <number:text>-</number:text>
    <number:currency-symbol number:language="en"
        number:country="US">$</number:currency-symbol>
    <number:number number:decimal-places="2"
```

```
        number:min-integer-digits="1" number:grouping="true"/>
    <style:map style:condition="value()&gt;=0"
        style:apply-style-name="N104P0"/>
</number:currency-style>
<style:style style:name="Result2" style:family="table-cell"
   style:parent-style-name="Result"
   style:data-style-name="N104" />
```

Example 2.6. Style Defintion in a Drawing Document

```
<style:style style:name="objectwitharrow"
    style:family="graphic" style:parent-style-name="standard">
    <style:graphic-properties
        draw:stroke="solid"
        svg:stroke-width="0.15cm" svg:stroke-color="#000000"
        draw:marker-start="Arrow" draw:marker-start-width="0.7cm"
        draw:marker-start-center="true"
        draw:marker-end-width="0.3cm"/>
</style:style>
```

The content.xml File

Although the details of the `content.xml` vary widely depending upon the type of document you are dealing with, there are elements which are common to all `content.xml` files. The root element is the `<office:document-content>` element. It defines all the namespaces that will be used throughout the document. The `office:version` attribute tells you which version of OpenDocument was used in the document.

The following elements are contained within the `<office:document-content>` element. The optional `<office:scripts>` element does appear in most documents and is always empty, even if your document contains macros. Go figure.

The `<office:scripts>` is followed by elements that describe the document's presentation. The optional `<office:font-face-decls>` element describes fonts used in your document, and duplicates the information found in `styles.xml`. If you have defined any styles "on the fly," then these automatic styles are described in the optional `<office:automatic-styles>` element.

The last child element of `<office:document-content>` is the required, and all-important, `<office:body>` element. This is where all the action is, and we will spend much of the rest of this book examining its contents. Its first child element tells which kind of document we are dealing with:

```
<office:text>
<office:drawing>
<office:presentation>
<office:spreadsheet>
<office:chart>
<office:image>
```

Example 2.7, “Structure of the content.xml file” shows the skeleton for an OpenOffice.org document’s `content.xml` file.

Example 2.7. Structure of the content.xml file

```
<office:document-content namespace declarations
  office:version="1.0"
  office:class="document type">

    <office:scripts/>

    <office:font-face-decls>
        <!-- font specifications -->
    </office:font-decls>

    <office:styles>
        <office:automatic-styles>
            <!-- style information -->
        </office:automatic-styles>
    </office:styles>

    <office:body>
        <office:documentType>
            <!-- actual content here -->
        </office:documentType>
    </office:body>
</office:document-content>
```

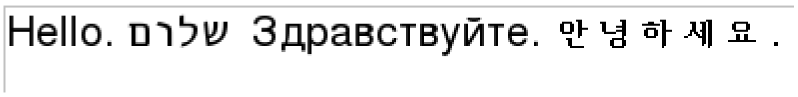
Chapter 3. Text Document Basics

At this point we are ready to look at the specifics of the `content.xml` file for word processing documents. We will build up from the most basic elements, characters and paragraphs, to sections and pages. This chapter also covers the topic of lists and outlines in OpenDocument word processing documents.

Characters and Paragraphs

All OpenDocument files are based on Unicode, and are encoded in the UTF-8 encoding scheme. You may see a discussion of this at the section called “Unicode Encoding Schemes”. This means that you may freely mix characters from a variety of languages in an OpenDocument file, as shown in Figure 3.1, “Document with Mixed Languages”. It also means that those characters will not be easily viewable in a normal ASCII text editor.

Figure 3.1. Document with Mixed Languages



Hello. שלום Здравствуйте. 안 녕 하 세 요 .

Whitespace

In XML, whitespace in element content is typically not preserved unless specially designated. OpenDocument collapses consecutive whitespace characters, which are defined as space (`0x0020`), tab (`0x0009`), carriage return (`0x000D`), and line feed (`0x000A`) to a single space. How, then, does OpenDocument represent a document where whitespace is significant?

To handle extra spaces, OpenDocument uses the `<text:s>` element. This empty element has an optional attribute, `text:c`, which tells how many spaces occur in the document. If this attribute is absent, then the element inserts one space. Between words, the `<text:s>` element is used to describe spaces *after* the first one; thus, for a single space, you don’t need this element. At the beginning of a line, you do need the `<text:s>`, since OpenDocument eliminates leading whitespace immediately after a starting tag.

Tab stops are represented by the empty `<text:tab>` element, and a line break, which is entered in OpenOffice.org by pressing Shift-Enter, is represented by the empty `<text:line-break>` element. Example 3.1, “Representation of Whitespace” shows these elements in action.

Example 3.1. Representation of Whitespace

```
<!--
   The following is the XML for
   .Hello,..whitespace!...  (where . represents the spacebar)
   Hello,--tab stops!       (where - represents the Tab key)
   Hello,|line break!       (where | represents Shift-Enter)
-->
<text:s/>Hello, <text:s/>whitespace! <text:s text:c="2"/>
Hello,<text:tab/><text:tab/>tab stops!
Hello,<text:line-break/>line break!
```

If you are using XSLT to extract the contents of an OpenDocument file to a plain text file, you may want to expand these elements into their original whitespace. Example 3.2, "XSLT Templates for Expanding Whitespace" shows the XSLT templates required to do this. The templates for `<text:tab>` and `<text:line-break>` are easy; we just emit the proper Unicode value. The code becomes slightly complex when we get to `<text:s>` because we need to be able to handle an arbitrary number of spaces. Here's the pseudocode:

- Create a variable named `spaces`, which contains 30 spaces. Remember to use the `xml:space="preserve"` attribute to prevent Xalan from "helpfully" collapsing this whitespace.
- If the `<text:s>` doesn't have a `text:c` attribute, simply emit one blank.
- If there is a `text:c` attribute, call a template named `insert-spaces` and pass the number of spaces in as a parameter named `n`.
- `insert-spaces` tests to see if `$n` is less than or equal to 30. If so, then the template emits that many spaces as a substring from the `$spaces` variable.
- If there are more than 30 spaces required, `insert-spaces` emits the entire `$spaces` variable, and then calls itself with `$n` minus 30 as the new number of spaces to emit.

[This is file `uncompress_whitespace.xsl` in directory `ch03` in the downloadable example files.]

Example 3.2. XSLT Templates for Expanding Whitespace

```
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns:text="urn:oasis:names:tc:opendocument:xmlns:text:1.0">

<xsl:template match="text:tab">
<xsl:text>&#x09;</xsl:text> <!-- emit a Unicode tab -->
</xsl:template>

<xsl:template match="text:line-break">
<xsl:text>&#x0a;</xsl:text> <!-- emit a Unicode line feed -->
</xsl:template>

<!-- there are 30 spaces after the opening tag -->
<xsl:variable name="spaces"
```

```
    xml:space="preserve">
</xsl:variable>

<xsl:template match="text:s">
<xsl:choose>
    <xsl:when test="@text:c">
        <xsl:call-template name="insert-spaces">
            <xsl:with-param name="n" select="@text:c"/>
        </xsl:call-template>
    </xsl:when>
    <xsl:otherwise>
        <xsl:text> </xsl:text>
    </xsl:otherwise>
</xsl:choose>
</xsl:template>

<xsl:template name="insert-spaces">
<xsl:param name="n"/>
<xsl:choose>
    <xsl:when test="$n &lt;= 30">
        <xsl:value-of select="substring($spaces, 1, $n)"/>
    </xsl:when>

    <xsl:otherwise>
        <xsl:value-of select="$spaces"/>
        <xsl:call-template name="insert-spaces">
            <xsl:with-param name="n">
                <xsl:value-of select="$n - 30"/>
            </xsl:with-param>
        </xsl:call-template>
    </xsl:otherwise>
</xsl:choose>
</xsl:template>
</xsl:stylesheet>
```

If you are creating an OpenDocument file from a source where whitespace has been preserved, you must reverse this process, creating the appropriate `<text:s>`, `<text:tab>`, and `<text:line-break>` elements. While there may be a simple and clever way of doing this conversion in XSLT, it eludes this author entirely. The straightforward approach of looking at the input string character-by-character is totally unsuited to the XSLT processing model, so we have created a Java extension function, which you may find in the section called "OpenDocument White Space Representation". Example 3.3, "Test XML file for Whitespace Conversion" shows a section of a test XML file, and Example 3.4, "Test XSL file for Whitespace Conversion" shows part of the XSLT that calls the extension function. [The example files are named `whitespace_compress_test.xml` and `compress_whitespace.xsl` in directory `ch03` in the downloadable example files.]

Example 3.3. Test XML file for Whitespace Conversion

```
<document xml:space="preserve">
<test> One leading blank; one trailing. </test>
<test>  Two leading blanks; two trailing.  </test>
<test>No leading blanks;    four internal; no trailing.</test>
```

```
<test>Two       tab stops.</test>

<test>Two
line
breaks.</test>
</document>
```

Example 3.4. Test XSL file for Whitespace Conversion

```
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns:text="urn:oasis:names:tc:opendocument:xmlns:text:1.0"
    xmlns:java="ODWhiteSpace" ❶
    exclude-result-prefixes="java">

<xsl:template match="/document">
<office:document-content
    xmlns:office="urn:oasis:names:tc:opendocument:xmlns:office:1.0" ❷
    xmlns:text="urn:oasis:names:tc:opendocument:xmlns:text:1.0"
    office:version="1.0">
    <office:body>
        <office:text>
    <xsl:apply-templates select="test"/>
        </office:text>
    </office:body>
</office:document-content>
</xsl:template>

<xsl:template match="test">
<xsl:variable name="str" select="."/>
<text:p><xsl:copy-of
    select="java:compressString($str)"/></text:p> ❸
</xsl:template>

</xsl:stylesheet>
```

❶ We are using the abbreviated format for Xalan extensions written in Java. The `xmlns:java` describes the path name to the extension. We have placed the `ODWhiteSpace.class` file in the same directory as the transformation program, so the fully qualified class name is just the class name; the `exclude-result-prefixes` ensures that the `java` prefix doesn't appear in the transformation output.

❷ This provides the bare bones for the document we are creating; we are declaring only the namespaces that are required in the resulting document.

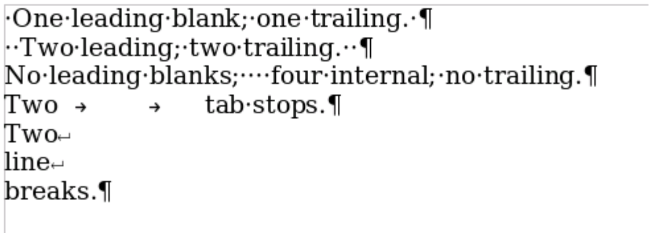
❸ When you call the extension function, it returns a set of nodes that represent your input string in OpenDocument format. If you use `<xsl:copy-of>`, the entire set will be copied to the output document. Don't use `<xsl:value-of>`; that will convert the entire node set to a string, and you'll get the string value of only the first node.

To run this transformation with the program shown in the section called “An XSLT Transformation”, we use the following command line, which invokes the shell script from the section called “Transformation Script”:

```
odtransform.sh -in whitespace_compress_test.xml \
  -xsl compress_whitespace.xsl \
  -outOD whitespace_compress_test.odt -out content.xml
```

The product is a file named `whitespace_compress_test.odt`. Figure 3.2, “Document Created with Whitespace XSLT Extension” shows the result; the font has been made larger, and non-printing characters are displayed so that we can check that the file has the correct content.

Figure 3.2. Document Created with Whitespace XSLT Extension



```
·One·leading·blank;·one·trailing.·¶
··Two·leading;·two·trailing.··¶
No·leading·blanks;····four·internal;·no·trailing.¶
Two →      →     tab·stops.¶
Two↵
line↵
breaks.¶
```

Before proceeding, let’s note two things about the preceding example. First, congratulations! We’ve just created our first OpenDocument file without using an application like OpenOffice.org. Second, we sneaked the `<text:p>` element into the example.

Defining Paragraphs and Headings

Almost all the text in a document will be enclosed either in paragraphs or headings. The `<text:p>` and `<text:h>` elements provide these functions. The `<text:h>` element has a `text:level` attribute; its integer value starts at one to signify a level one heading. If you’re extracting data from an OpenDocument file, that’s really all you need to know about these two elements.

Note

If you use the non-numbered `Heading` style from the stylist, OpenOffice.org will insert a `<text:p>`, not a `<text:h>` element.

Character and Paragraph Styles

If you’re creating an OpenDocument file by using an XSLT transformation, you don’t want it to say “I am a Fugitive from a Chain Printer.” You will need to add styles to your document, and this will require three steps:

1. Create font declarations by putting `<style:font-decl>` elements within an `<office:font-decls>` element. If you just want to use the default font everywhere, this step is optional. (Even if you do want to use a variety of fonts, you can skip this step. We'll show you the workaround later, and give you a stern warning not to use it.)
2. Create `<style:style>` elements within an `<office:automatic-styles>` element. Each of these will contain a `<style:`*`type`*`-properties>` element whose attributes describe the style you want.
3. Refer to the styles created in step two with the `text:style-name` attribute.

Note

Throughout this book, we will be dealing with the concept of a *length*. In stylesheets, a *length* is expressed as a number followed immediately by a unit of measurement. For example, `3.05cm` or `24pt`. The valid units in OpenDocument are `in`, `cm`, `mm`, `px` (pixels), `pc` (picas; 6 picas equals one inch), and `pt` (points; 72 points equal one inch). These *lengths* are actually a general way of measuring things. Thus, you will see sentences like, "The `width` attribute has a value which is a *length*." That means you might write something like `width="2.57mm"`. Whenever we use the word "*length*" in this generic sense, it will be set in italics.

Creating Font Declarations

Font declarations are written as described in the section called "Font Declarations". Rather than writing them yourself, you may wish to use an existing document, or you may create a document in an OpenDocument-compatible application which contains one letter from each font you will want. You can then unpack the files and copy the declarations. Your third option is to write them by hand.

No matter which method you use, we recommend you put the resulting declarations in a separate file and include them with an `<xsl:include>`. This makes your primary stylesheet shorter, and it also allows re-use of the declarations in other transformations. Example 3.5, "Font Declarations Include File" shows a sample stylesheet for inclusion into your primary transformation; Example 3.6, "Using an Included Font Declaration File" shows how you would use it.

Example 3.5. Font Declarations Include File

```
<!-- save this in file fontdecls.xslt -->
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns:office="urn:oasis:names:tc:opendocument:xmlns:office:1.0"
    xmlns:style="urn:oasis:names:tc:opendocument:xmlns:style:1.0"
    xmlns:svg="urn:oasis:names:tc:opendocument:xmlns:►
svg-compatible:1.0"
    xmlns:fo="urn:oasis:names:tc:opendocument:xmlns:►
xsl-fo-compatible:1.0">
```

```
<xsl:template name="insert-font-decls">
    <office:font-face-decls>
        <style:font-face style:name="Arial Unicode MS"
            svg:font-family="'Arial Unicode MS'"
        style:font-pitch="variable" />
        <style:font-face style:name="Bitstream Vera Sans"
            svg:font-family="&apos;Bitstream Vera Sans&apos;"
            style:font-family-generic="swiss"
            style:font-pitch="variable"/>
        <style:font-face style:name="Bitstream Vera Sans1"
            svg:font-family="&apos;Bitstream Vera Sans&apos;"
            style:font-pitch="variable"/>
        <style:font-face style:name="Bitstream Vera Serif"
            svg:font-family="&apos;Bitstream Vera Serif&apos;"
            style:font-family-generic="roman"
            style:font-pitch="variable"/>
         <style:font-face style:name="Lucidasans"
            svg:font-family="Lucidasans"
            style:font-pitch="variable"/>
        <style:font-face style:name="Palatino"
            svg:font-family="Palatino"
            style:font-family-generic="roman"
            style:font-pitch="variable" />
        <style:font-face style:name="Bitstream Vera Sans Mono"
            svg:font-family="&apos;Bitstream Vera Sans Mono&apos;"
            style:font-pitch="fixed"/>
    </office:font-face-decls>
</xsl:template>
</xsl:stylesheet>
```

Example 3.6. Using an Included Font Declaration File

```
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns:office="urn:oasis:names:tc:opendocument:xmlns:office:1.0"
    xmlns:style="urn:oasis:names:tc:opendocument:xmlns:style:1.0"
    xmlns:fo="urn:oasis:names:tc:opendocument:xmlns:►
xsl-fo-compatible:1.0"
    xmlns:text="urn:oasis:names:tc:opendocument:xmlns:text:1.0"
> <!-- and any other namespaces you might require -->

<xsl:include href="fontdecls.xsl"/>

<xsl:template match="/">
<office:document-content
    xmlns:office="urn:oasis:names:tc:opendocument:xmlns:office:1.0"
    xmlns:style="urn:oasis:names:tc:opendocument:xmlns:style:1.0"

    xmlns:fo="urn:oasis:names:tc:opendocument:xmlns:►
xsl-fo-compatible:1.0"
    xmlns:text="urn:oasis:names:tc:opendocument:xmlns:text:1.0">
<!-- and any other namespaces you might require -->
    <office:scripts/>
    <xsl:call-template name="insert-font-decls"/>
    <!-- remainder of your transformation -->
</office:document-content>
</xsl:template>

</xsl:stylesheet>
```

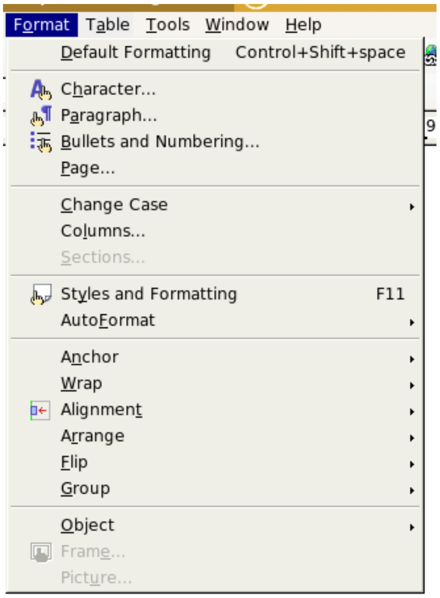
Creating Automatic Styles

Your transformation will then create an `<office:automatic-styles>` element, which will hold all the `<style:style>` elements that you want. You should give each `<style:style>` an appropriate `style:name` and `style:family` attribute. For OpenOffice.org documents, styles to be applied to characters should have a `style:name` of the form `T` followed by an integer and have a `style:family` of `text`. Styles to be applied to paragraphs or headings should have a `style:name` of the form `P` followed by an integer and have a `style:family` of `paragraph`.

Character Styles

OpenDocument applications such as OpenOffice.org let you change the format of individual characters, paragraphs, or pages, as you see in Figure 3.3, “OpenOffice.org Format Menu”. The following attributes of the `<style:text-properties>` element affect character styles. Most of these attributes come from the XSL-FO namespace. (The `<style:text-properties>` element will be contained in a `<style:style>` element.)

Figure 3.3. OpenOffice.org Format Menu



`style:font-name`
: The name of a `<style:font-face-decl>`. If you do not have any font declarations, you may use a `svg:font-family` attribute with a font name as its value. This is cheating. Don’t do it. (We said we would warn you!)

`fo:font-size`
: The text size, expressed either as a *length* or a percentage. For fonts, a *length* is expressed as a positive integer followed by `pt` (points). Other units of measurement will be converted to points when you view them in OpenOffice.org.

`fo:font-weight`
: Values are `bold` and `normal`.

`fo:font-style`
: Values are `italic` and `normal`.

`style:text-underline-style`, `style:text-underline-type`, `style:text-underline-color`, `style:text-underline-width`, `style:text-underline-mode`
: *Oy*, you wouldn't believe how many combinations of underlining you have available to you! The `style:text-underline-style` attribute may have values `none`, `solid`, `dotted`, `dash`, `long-dash`, `dot-dash`, `dot-dot-dash`, and `wave`.

  The `style:text-underline-type` has the possible values `none`, `single`, and `double`. The default is `single`.

  The `style:text-underline-width` has values `auto`, `normal`, `bold`, `thin`, `dash`, `medium`, and `thick`.

  The `style:text-underline-color` is specified as in `fo:color` and has the additional value of `font-color`, which makes the underline color the same as the current text color.

  Finally, `style:text-underline-mode` may be set to `skip-white-space` so that whitespace is not underlined.

`fo:color`, `fo:background-color`
: Text color and background color in the form of a six-digit hex value. Example: `#cc32f5`. The background color may also be set to `transparent`.

`fo:font-variant`
: This can have a value of `normal` or `small-caps`.

`fo:text-transform`
: Possible values are `none`, `lowercase`, `uppercase`, `capitalize`, and `small-caps`. `capitalize` corresponds to the “Title” choice in OpenOffice.org's Character Font Effects dialog, which capitalizes the first letter of every word. `uppercase` corresponds to the “Capitalize” choice, which displays all the words in uppercase.

`style:text-outline`
: Set to `true` to get outline text

`fo:text-shadow`
: The value is the horizontal and vertical displacement of the shadow, separated by whitespace. The default value is `1pt`.

`style:text-blinking`
: Set to `true` if you want the readers of your document to hate you forever.

`style:text-position`
: This attribute is used to create superscripts and subscripts. It can have two values; the first value is either `sub` or `super`, or a number which is the percentage of the line height (positive for superscripts, negative for subscripts). An optional second value gives the text height as a percentage of the current font height. You must provide the percent symbol. Examples: `style:text-position="super"` produces normal superscripts, and `style:text-position="-30% 50%"` produces a subscript at 30% of the font height below the baseline, with letters 50% of the current font height.

`style:text-rotation-angle`, `style:text-rotation-scale`
: The angle is the number of degrees to rotate text counterclockwise; its value can be 0, 90, or 270. The text rotation scale can be set to `line-height` or `fixed`.

`style:text-scale`
: The percentage by which to scale character width, given with a percent symbol; for example: `50%`.

`fo:letter-spacing`
: The value is a *length* to add to the normal inter-character spacing; it can be positive to expand text or negative to compress text. If you want kerning of double characters, set `style:letter-kerning` to `true`.

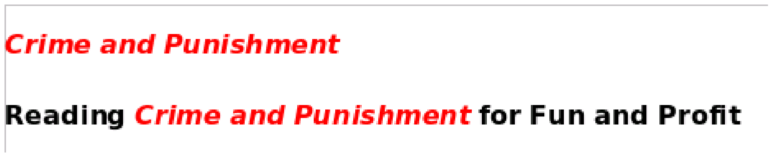
`style:font-relief`
: Possible values are `none`, `embossed`, and `engraved`.

Using Character Styles

Before we go further, let’s put these to work. Figure 3.4, “Styled Headings” shows two headings. The first one is a level five heading which we have made red and italic. (If you are reading this in a printed book, use your imagination to see the color.) The second heading is a level five heading with the red italics applied to only some of the words. In order to apply styles to only part of a paragraph or heading, we need to enclose it in a `<text:span>` element, which delineates an inline area of text.

In any case, we do *not* apply the style attributes directly to the heading, paragraph, or span. Instead, we declare the style in the `<office:automatic-styles>` area and then use a `text:style-name` attribute in the `<text:p>`, `<text:h>`, or `<text:span>`.

Figure 3.4. Styled Headings



Example 3.7, “Markup for <text:h>” shows the relevant excerpts of the XML for the two headings.

Example 3.7. Markup for <text:h>

```
<!-- from styles.xml -->
<style:style style:family="paragraph"
    style:name="Heading_20_5" style:display-name="Heading 5" ❶
    style:parent-style-name="Heading" style:class="text"
    style:default-outline-level="5">
    <style:text-properties fo:font-size="85%" fo:font-weight="bold"/>❷
</style:style>

<!-- from content.xml -->
<office:automatic-styles>
<style:style style:name="P1" style:family="paragraph" ❸
    style:parent-style-name="Heading_20_5">
   <style:text-properties fo:color="#ff0000"
    fo:font-size="11.8500003814697pt"
    fo:font-style="oblique" /> ❹

<style:style style:name="T1" style:family="text"> ❺
    <style:text-properties fo:color="#ff0000" fo:font-style="italic"
/>
</style:style>
</office:automatic-styles>

<office:body>
<text:h text:style-name="P1" text:outline-level="5">Crime and ❻
    Punishment</text:h>
<text:h text:style-name="Heading 5" text:level="5">Reading
        <text:span text:style-name="T1">Crime and
        Punishment</text:span> for Fun and Profit</text:h>
</office:body>
```

❶ The level 5 heading style is found in the `styles.xml` file; its `style:family` attribute shows that it applies to block elements such as paragraphs and headings. Note the `style:display-name`, which shows the style name without the hexadecimal encoding of the blank as `_20_`.

❷ The styles that apply to the characters in the heading are stored in a `<style:text-properties>` element.

❸ This definition in `content.xml` creates a style based on `Heading 5`, so it is also has `style:family="paragraph"` and its `style:name` begins with `P`.

❹ This style sets `fo:font-style` to `oblique` instead of `italic`, for some unknown reason.

❺ The inline style has a `style:family="text"` and its `style:name` begins with `T`.

❻ Styles are always applied by referring to the appropriate `text:style-name`.

Paragraph Styles

Paragraph styles affect the location, indention, and look of paragraphs (and headings). Here are some of the styles you will most commonly use.

`fo:line-height`
: This specifies a fixed line height; specifying `none` does its normal line height calculation. Specifying a *length* (`24pt`) or a percentage (`125%`) may lead to overlapping or cut-off text if some characters are larger than the line height.

`style:line-height-at-least`
: The value is a *length* which specifies the minimum line height.

`style:line-spacing`
: The value is a *length* that specifies a fixed distance between lines in a paragraph.[2]

`fo:text-align`, `fo:text-align-last`
: Values for this attribute are `start`, `end`, `center`, and `justify`. OpenOffice.org maps "left" to `start` and "right" to `end`, no matter the directionality of the text.

  If you choose justified text, then you have the option to position the last line as well by setting `fo:text-align-last` to `justify`, `center`, or `start`.

`fo:margin-left`, `fo:margin-right`, `fo:text-indent`
: The values for the margins are a positive *length* telling how far to indent from the given side; the value for the first-line indent can be either positive or negative. If you specify `fo:text-indent`, you must also specify margins.

[2] `fo:line-height`, `style:line-height-at-least`, and `style:line-spacing` are mutually exclusive.

`fo:margin-top`, `fo:margin-bottom`
: These attributes control spacing before and after a paragraph. The value for these attributes is a *length* or a percentage relative to the parent style.

`fo:break-before`, `fo:break-after`
: The value of `column` or `page` tells whether to put a column or page break before or after the paragraph. You may use only one of these in a style specification. The default value of `auto` lets the application make the decision as to whether a break is necessary before the text.

`fo:orphans`, `fo:widows`
: A number giving the minimum number of lines in a paragraph before and after a page break. You may keep a paragraph together with the next one by setting `fo:keep-with-next` to `always`.

`fo:background-color`
: This is the background color for the paragraph, expressed as a six-digit hex value.

Borders and Padding

You may draw borders on all four sides of a paragraph by specifying `fo:border`. You may set an individual side with `fo:border-left`, `fo:border-right`, `fo:border-top`, and `fo:border-bottom`. Each of these has a value of the form:

`width style color`

Where:

- *width* is a *length* specification or one of the keywords `thick` and `thin`.
- *style* is one of `none`, `solid`, or `double`.
- *color* is a six-digit hexadecimal color value.

If you have a `double` border, you may completely control the spacing of the lines by specifying a `style:border-line-width` (or `style:border-line-width-`*`side`*) attribute which has three *length* specifiers as its value:

- The width of the inner line.
- The space between the lines.
- The width of the outer line.

Example 3.8, "Border Specification" shows the markup required for a green double border on all four sides, with an inner line width of 0.5 millimeter, a distance of 0.25 millimeters between the lines, and an outer line width of 1 millimeter. The total width of the border is the sum of the individual widths and distances of the `style:border-line-width` attribute.

Example 3.8. Border Specification

```
<style:style style:name="P1" style:family="paragraph">
    <style:properties
        style:border-line-width="0.5mm 0.25mm 1mm"
        fo:border="1.75mm double #008000"/>
</style:style>
```

To set the padding between the border and the paragraph content, use the `fo:padding` attribute (for all four sides), or `fo:padding-left`, `fo:padding-right`, `fo:padding-top`, and `fo:padding-bottom` to set padding on sides individually. The value of these attributes is a *length* specifier.

> **Note**
> Padding will not be shown unless there is a border. For example, if you set `fo:padding-left` but do not have a `fo:border-left` or `fo:border`, OpenOffice.org will not add any padding on the left.

Tab Stops

So, how does the `<text:tab>` element know where the tabs are? You tell it by adding a `<style:tab-stops>` element, which contains a list of `<style:tab-stop>` elements.

The `<style:tab-stop>` element has a required `style:position` attribute, whose value is a *length* specification. By default, tab stops are left-aligned, but you may change this with the `style:type` attribute, whose value may be one of `left`, `center`, `right`, or `char`. This last value lets you align on a specific character, such as a decimal point. The character on which to align is specified as the value of the `style:char`. Space between tab stops is normally filled with blanks. You may specify a different filler character (also called a “leader”) as the value of the `style:leader-text` attribute. The leader character fills the space *before* the tab stop. You may also specify a `style:leader-style` with a value of `solid`, `dotted`, or `dash`.

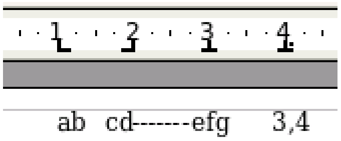
Example 3.9, “Various Tab Stops” shows the XML for a paragraph with a left-aligned tab stop at one centimeter, a right-aligned stop at two centimeters, a centered stop at three centimeters with a dash as a leader character, and a tab stop on comma at four centimeters. Figure 3.5, “Tab Stops in OpenOffice.org” shows a paragraph using this formatting.

Example 3.9. Various Tab Stops

```
<style:style style:name="P3" style:family="paragraph">
    <style:paragraph-properties>
        <style:tab-stops>
            <style:tab-stop style:position="1cm"/>
            <style:tab-stop style:position="2cm" style:type="right"/>
            <style:tab-stop style:position="3cm" style:type="center"
```

```
                style:leader-style="dash" style:leader-text="-"/>
            <style:tab-stop style:position="4cm" style:type="char"
                style:char=","/>
        </style:tab-stops>
    </style:paragraph-properties>
    <style:text-properties style:font-name="Bitstream Charter"/>
</style:style>
```

Figure 3.5. Tab Stops in OpenOffice.org



Asian and Complex Text Layout Characters

Not all documents are written in the Latin alphabet. There is the entire family of CJK (Chinese-Japanese-Korean) syllables and ideographs. Languages which do not lay out their characters in a simple left-to-right fashion are in a class called Complex Text Layout; examples of such languages are Arabic, Hebrew, Hindi, and Thai. OpenDocument lets you specify certain character attributes for the CJK characters by adding the `-asian` suffix; attributes for Complex Text Layout characters have a suffix of `-complex`. In the following style, Western and Complex characters will be bold, but CJK characters will be normal:

```
<style:style style:name="T5" style:family="text">
    <style:text-properties
        fo:font-weight="bold"
        style:font-weight-asian="normal"
        style:font-weight-complex="bold" />
</style:style>
```

Here is a list of the attributes which can take either the `-asian` or `-complex` suffix:

`style:font-name-asian`, `style:font-name-complex`
`style:font-name-asian`, `style:font-name-complex`
`style:font-family-asian`, `style:font-family-complex`
`style:font-family-generic-asian`, `style:font-family-generic-complex`
`style:font-style-name-asian`, `style:font-style-name-complex`
`style:font-pitch-asian`, `style:font-pitch-complex`
`style:font-size-asian`, `style:font-size-complex`
`style:font-size-rel-asian`, `style:font-size-rel-complex`
`style:language-asian`, `style:language-complex`
`style:country-asian`, `style:country-complex`
`style:font-style-asian`, `style:font-style-complex`
`style:font-weight-asian`, `style:font-weight-complex`

Case Study: Extracting Headings

At this point, we know enough to write an XSLT document that will extract all the headings from an OpenDocument file and create a new “outline” document. A heading at level *x* is preceded by *x*-1 tabs. Thus, a level one heading starts in the left margin of the document, and a level three heading has two tab stops followed by the heading text.

Example 3.10. Extracting Headings from an OpenDocument File

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"  ❶
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns:office="urn:oasis:names:tc:opendocument:xmlns:office:1.0"
    xmlns:script="urn:oasis:names:tc:opendocument:xmlns:script:1.0"
    xmlns:style="urn:oasis:names:tc:opendocument:xmlns:style:1.0"
    xmlns:text="urn:oasis:names:tc:opendocument:xmlns:text:1.0"
    xmlns:fo="urn:oasis:names:tc:opendocument:xmlns:►
xsl-fo-compatible:1.0"
>

<xsl:template match="/office:document-content">
    <office:document-content
      xmlns:office="urn:oasis:names:tc:opendocument:xmlns:office:1.0"
      xmlns:script="urn:oasis:names:tc:opendocument:xmlns:script:1.0"
      xmlns:style="urn:oasis:names:tc:opendocument:xmlns:style:1.0"
      xmlns:text="urn:oasis:names:tc:opendocument:xmlns:text:1.0"
    >
    <office:scripts/>

        <office:automatic-styles>
            <style:style style:name="P1" style:family="paragraph">
            <style:paragraph-properties> ❷
                <style:tab-stops>
                    <style:tab-stop style:position="1cm" />
                    <style:tab-stop style:position="2cm" />
                    <style:tab-stop style:position="3cm" />
                    <style:tab-stop style:position="4cm" />
                    <style:tab-stop style:position="5cm" />
                    <style:tab-stop style:position="6cm" />
                    <style:tab-stop style:position="7cm" />
                </style:tab-stops>
            </style:paragraph-properties>
            <style:text-properties
                     fo:font-size="10pt"
                     style:font-size-asian="12pt"
                     style:font-size-complex="12pt"/>
            </style:style>
        </office:automatic-styles>
        <office:body>
            <office:text>
                <xsl:apply-templates select="//text:h"/> ❸
            </office:text>
        </office:body>
        </office:document-content>
</xsl:template>
```

```
<xsl:template match="text:h">
    <text:p text:style-name="P1">
        <xsl:call-template name="emit-tabs">
            <xsl:with-param name="n"
               select="@text:outline-level - 1"/>
        </xsl:call-template>
    <xsl:value-of select="."/>
    </text:p>
</xsl:template>

<xsl:template name="emit-tabs"> ❹
    <xsl:param name="n" select="0"/>
    <xsl:if test="$n != 0">
        <text:tab/>
        <xsl:call-template name="emit-tabs">
            <xsl:with-param name="n" select="$n - 1"/>
        </xsl:call-template>
    </xsl:if>
</xsl:template>
</xsl:stylesheet>
```

❶ To make this document shorter, we've included only the namespaces that are necessary both here and in the root element of the output document.

❷ There are two ways to implement this stylesheet. You can create a separate style for each level of heading, or have a single style with all the tab stops in it. The second method seems more to the point, though it does make the processing a bit more difficult.

❸ This is the place where laziness triumphs; the XPath expression finds all `<text:h>` elements at any level in the document. It's inefficient in terms of machine time, but easy for us.

❹ Karma strikes back; because we were lazy earlier, we have to do some hard work here. XSLT doesn't have a "for loop" such as those in procedural programming languages, so we need to produce the required number of `<text:tab>` elements by recursion. This template gets a parameter, `n`; if the value is non-zero, the template emits a `<text:tab-stop>` into the output, then calls itself to emit `n-1` more tab stops.

You will find a sample file, XSLT file, and resulting heading file in files `document_with_headings_to_extract.odt`, `extract_headings.xsl`, and `extracted_headings.odt` in the `ch03` directory in the downloadable example files. The command to do the transform was:

```
odtransform.sh -inOD document_with_headings_to_extract.odt \
 -in content.xml \
 -xsl extract_headings.xsl \
 -out content.xml -outOD extracted_headings.odt
```

Sections

You start a new section of a document by enclosing your content in a `<text:section>` element. This element has a required `text:name` attribute, which has the same name as an existing `<style:style>` element. The `<text:section>` element also has an optional `text:style-name` attribute, whose value is an internal name for the section.

The `<style:style>` that the section refers to will have a `style:family="section"`. As with all other `<style:style>` elements, it will contain a `<style:section-properties>` element. If the contents of the columns are to be evenly distributed to all the columns, then the `<style:section-properties>` element will have its `text:dont-balance-text-columns` attribute set to `false`. A value of `true` indicates that column contents are not to be distributed equally.

The `<style:section-properties>` element will in turn contain a `<style:columns>` element. The `<style:columns>` element has a required `fo:column-count` attribute, whose value is the number of columns. A one-column section has a column count of zero. The `fo:column-gap` attribute gives the spacing between the columns.

The `<style:columns>` element contains one `<style:column>` element for each column. Each column has these attributes:

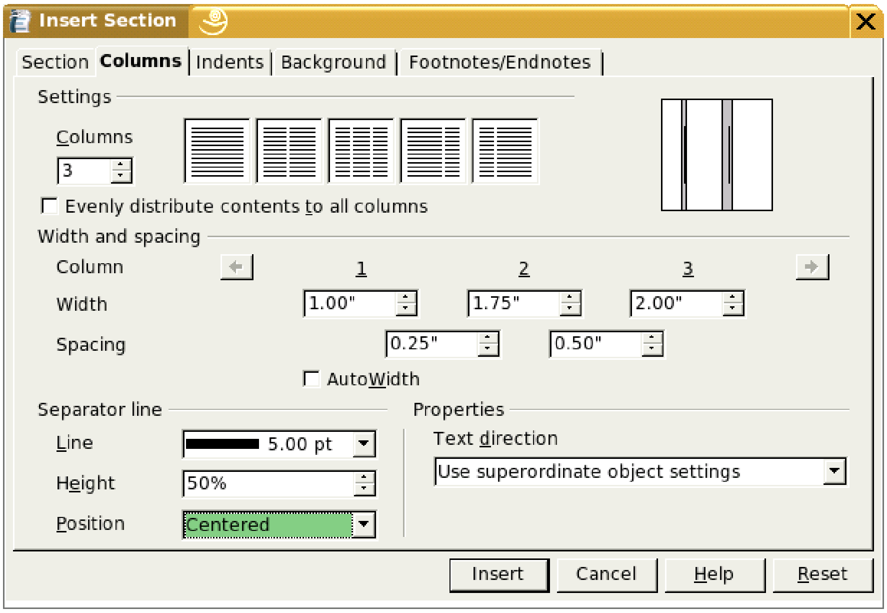
`style:rel-width`
: The proportional width of the column expressed in twips[3] followed by an asterisk instead of a *length* unit. Thus, a one-inch wide column is specified as `style:rel-width="1440*"`.

`fo:start-indent`, `fo:end-indent`
: These specify the inter-column spacing in absolute units.

The `style:rel-width` includes inter-column spacing. Given the specifications shown in Figure 3.6, “Column Spacing”, the total width of the first column is 1.125 inches (one inch plus half of the quarter-inch spacing). The total width of the second column is 2.125 inches (1.75 inches plus half of the quarter-inch spacing plus half of the half-inch spacing). The total width of the second column is 2.25 inches (two inches plus half of the half-inch spacing).

[3] A twip is one twentieth of a point, so there are 1440 twips per inch.

Figure 3.6. Column Spacing



Insert Section

Section | Columns | Indents | Background | Footnotes/Endnotes

Settings

Columns: 3

☐ Evenly distribute contents to all columns

Width and spacing

Column	1	2	3
Width	1.00"	1.75"	2.00"
Spacing	0.25"	0.50"	

☐ AutoWidth

Separator line

Line: 5.00 pt

Height: 50%

Position: Centered

Properties

Text direction: Use superordinate object settings

Insert | Cancel | Help | Reset

If you place separators between columns, then you must place a `<style:column-sep>` element before the first `<style:column>` element. The `<style:column-sep>` element has these attributes:

> `style:width`—the width of the separator line
> `style:color`— its color as a six-digit hex value
> `style:height`— the separator height as a percentage value
> `style:vertical-align`—either `top` (the default), `middle`, or `bottom` of the space between the columns.

Example 3.11, "OpenOffice.org Representation of Sections" shows the XML for the three-column section with column widths and spacing as shown in Figure 3.6, "Column Spacing". The columns have a five-point vertically centered separator line, and the section is indented one half inch from each margin. Content is not distributed evenly among the columns.

Example 3.11. OpenOffice.org Representation of Sections

```
<style:style style:name="Sect6" style:family="section">
    <style:section-properties text:dont-balance-text-columns="true"
        style:editable="false">
        <style:columns fo:column-count="3">
            <style:column-sep
                style:width="0.0693in"
                style:color="#000000"
                style:height="50%"
                style:vertical-align="middle"/>
            <style:column
                style:rel-width="1620*"
                fo:start-indent="0in"
                fo:end-indent="0.1252in"/>
            <style:column
                style:rel-width="3060*"
                fo:start-indent="0.1252in"
                fo:end-indent="0.25in"/>
            <style:column
                style:rel-width="3240*"
                fo:start-indent="0.25in"
                fo:end-indent="0in"/>
        </style:columns>
    </style:section-properties>
</style:style>
```

If you do not distribute content equally, then you may need to insert a manual section break within the text. This is done by applying a style with `fo:break-before="column"` in its `<style:paragraph-properties>`. Example 3.12, “Using a Section Style” shows the relevant style and content that uses the preceding section definition.

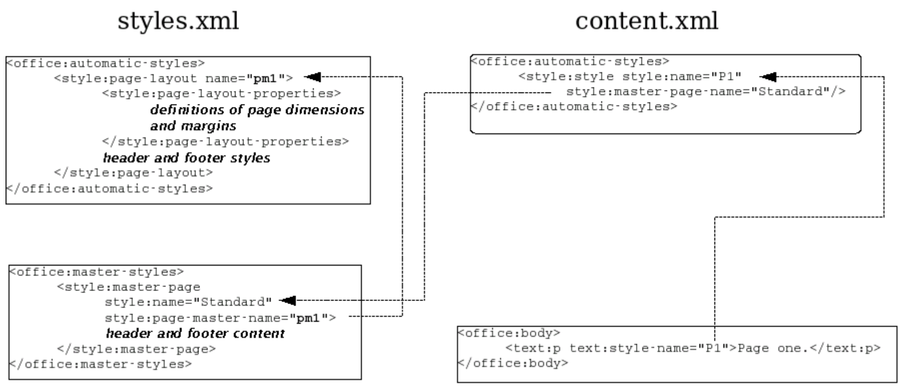
Example 3.12. Using a Section Style

```
<!-- This goes into the automatic-styles section -->

<style:style style:name="P1" style:family="paragraph"
    style:parent-style-name="Standard">
    <style:paragraph-properties fo:break-before="column" />
</style:style>

<!-- This is in the document content -->
<text:section text:style-name="Sect1" text:name="Section1">
    <text:p text:style-name="Standard">Column one</text:p>
    <text:p text:style-name="P1">Column two</text:p>
    <text:p text:style-name="P1">Column three</text:p>
</text:section>
```

Pages

Inserting a page break works in the same way as a section break: a paragraph or heading references a style which has a `<style:paragraph-properties>` element with a `fo:break-before="page"` attribute. Unlike sections, the specification for a page’s characteristics go into the `styles.xml` file rather than the `content.xml` file. However, the `content.xml` does refer to the style file, as shown in Figure 3.7, “Relationship Among Elements When Specifying Pages”.

Figure 3.7. Relationship Among Elements When Specifying Pages



```
<office:automatic-styles>
      <style:page-layout name="pm1">
            <style:page-layout-properties>
                  definitions of page dimensions
                  and margins
            </style:page-layout-properties>
            header and footer styles
      </style:page-layout>
</office:automatic-styles>
```

```
<office:master-styles>
      <style:master-page
            style:name="Standard"
            style:page-master-name="pm1">
            header and footer content
      </style:master-page>
</office:master-styles>
```

```
<office:automatic-styles>
      <style:style style:name="P1"
            style:master-page-name="Standard"/>
</office:automatic-styles>
```

```
<office:body>
      <text:p text:style-name="P1">Page one.</text:p>
</office:body>
```

Specifying a Page Master

The `styles.xml` file contains one `<style:page-layout>` element for every different type of page your document uses. This element has a required `style:name` attribute and an optional `style:page-usage` attribute. The page usage can have a value of `all` (the default), `left`, `right`, and `mirrored`. If you use `mirrored`, then margins are mirrored as you move from page to page. The `<style:page-layout>` elements are placed within the `<office:automatic-styles>` element.

The `<style:page-layout>`’s content starts with a `<style:page-layout-properties>` element that has these attributes:

`fo:page-width`, `fo:page-length`
: the value is a *length*, such as `21.9cm`.

`fo:margin-top`, `fo:margin-bottom`, `fo:margin-left`, `fo:margin-right`
: the value is a *length*.

`style:print-orientation`
: value is either `portrait` or `landscape`.

`style:writing-mode`
: one of: `lr-tb` (left to right; top to bottom), `rl-tb`, `tb-rl`, `tb-lr`, `lr`, `rl`, `tb`, and `page`. I have no clue what `page` value does.

`fo:background-color`
: a six-digit hex value, such as `#ffff99`; if omitted, the background is unfilled. If you are using a background image, then set `fo:background-color` to `transparent`, and use a `<style:background-image>` element as described in the section called “Background Images”.

`style:num-format`
: the page number format; possible values are `1`, the default of arabic numerals, `a` and `A` for lowercase and uppercase lettering, and `i` and `I` for lowercase and uppercase roman numerals.

`style:footnote-max-height`
: the value is a *length* giving the maximum footnote height. If the value is zero, then the footnote area cannot be larger than the page area. Note: although a value of zero does not require a unit specifier, it can be present, so you may see a value such as `0in`.

  If your document contains footnotes, then the `<style:properties>` element, in turn, contains an optional `<style:footnote-sep>` element that describes how footnotes are separated from the main text. Its attributes are:

`style:width`
: value is a *length* specifier giving the thickness of the separator line.

`style:distance-before-sep`, `style:distance-after-sep`
: these values are *length* specifiers that give the distance before and after the footnote separator (as their names so aptly indicate).

`style:adjustment`
: the alignment of the separator line; values are `left` (default), `center`, or `right`.

`style:rel-width`
: how far the separator extends across the page, expressed as a percentage. Thus, a separator that takes up one fourth of the page width has a value of `25%`.

But wait, that's not all. If you have headers and footers on your page, you must add a `<style:header-style>` and/or `<style:footer-style>` as appropriate. Each of these tags contains a `<style:header-footer-properties>` element that specifies any attributes you wish to apply to the header or footer. If you don't have a header or footer, then make these elements empty.

None of this is complicated; there's just so much of it. The following is an outline of what a page master style looks like:

```
<style:page-layout style:name="pm4">
    <style:page-layout-properties page width, height, margins, ►
writing mode>
        <style:footnote-sep line thickness, width, and distances />
    </style:page-layout-properties>

    <style:header-style>
        <style:header-footer-properties header specifications/>
    </style:header-style>
    <style:footer-style>
        <style:header-footer-properties footer specifications/>
    </style:footer-style>
</style:page-layout>
```

Example 3.13, “Full Page Master Specification” shows a complete specification for a landscape-oriented page that has both a header and footer. It has been reformatted for ease of reading. You may also wonder what is the absolute minimum you can get away with; if you are not using headers or footers, you can make a workable portrait-oriented page master with the specifications shown in Example 3.14, “Minimal Page Master Specification”.

Example 3.13. Full Page Master Specification

```
<style:page-layout style:name="pm2">
    <style:page-layout-properties
        fo:page-width="11in" fo:page-height="8.5in"
        style:num-format="1"
        style:print-orientation="landscape"
        fo:margin-top="0.5in" fo:margin-bottom="0.5in"
        fo:margin-left="0.5in" fo:margin-right="0.5in"
        style:writing-mode="lr-tb"
        style:footnote-max-height="0in">
        <style:footnote-sep style:width="0.0071in"
            style:distance-before-sep="0.0398in"
            style:distance-after-sep="0.0398in"
            style:adjustment="left"
            style:rel-width="25%"
            style:color="#000000"/>
    </style:page-layout-properties>
    <style:header-style>
        <style:header-footer-properties
            fo:min-height="0in" fo:margin-bottom="0.1965in"/>
    </style:header-style>
    <style:footer-style>
        <style:header-footer-properties
            fo:min-height="0in" fo:margin-top="0.1965in"/>
    </style:footer-style>
</style:page-layout>
```

Example 3.14. Minimal Page Master Specification

```
<style:page-master style:name="pm1">
    <style:page-layout-properties
        fo:margin-top="1in" fo:margin-bottom="1in"
        fo:margin-left="1.25in" fo:margin-right="1.25in" />
    <style:header-style />
    <style:footer-style />
</style:page-master>
```

Master Styles

In addition to the `<office:automatic-styles>` in the `styles.xml` file, you must have an `<office:master-styles>` element. This element contains header and footer content for each type of page, and also tells how pages follow one another. (For example, you might have a document where the first page is an envelope and the subsequent pages are letter-sized.)

The `<office:master-styles>` element contains one `<style:master-page>` for each `<style:page-master>` element you have defined. The `<style:master-page>` element has a required `style:name` attribute which gives the name that appears in the OpenOffice.org style catalog. The other required attribute is `style:page-layout-name`, whose value is the name of the `<style:page-layout>` defined earlier in the automatic styles.

If this page master has a specific page style that follows it (for example, a “title page” might be followed by a “contents page”), then you add a `style:next-style-name` attribute.

If your page has a header or footer, then this is where its content goes; *not* in the `content.xml` file where you might expect it. Example 3.15, “Master Styles” shows the master styles section for a document with two page styles, one of which is a landscape oriented page with a header and footer.

Example 3.15. Master Styles

```
<office:master-styles>
    <style:master-page style:name="Standard"
        style:page-master-name="pm1" />

    <style:master-page style:name="LandscapePage"
        style:page-master-name="pm2"
        style:next-style-name="Standard">
        <style:header>
            <text:p text:style-name="P1">Landscape ►
page header</text:p>
        </style:header>
        <style:footer>
            <text:p text:style-name="P2">Landscape page ►
footer</text:p>
        </style:footer>
    </style:master-page>

</office:master-styles>
```

If you have different headers (or footers) for left and right pages, then you may add a `<style:header-left>` or `<style:footer-left>` element to differentiate left pages from the (default) right pages.

Pages in the content.xml file

After all this activity in the `styles.xml` file, we can finally turn our attention to the `content.xml` file, where we actually put these specifications to use. The text at the beginning of the first of a series of pages references a `<style:style>` which has a `style:master-page-name` attribute that specifies the appropriate master page. Text at the beginning of subsequent pages references a style whose `<style:paragraph-properties>` element has a `fo:break-before="page"`. Example 3.16, "Using Master Pages in content.xml" shows the relevant portions of the `content.xml` for a simple document that has two pages.

Example 3.16. Using Master Pages in content.xml

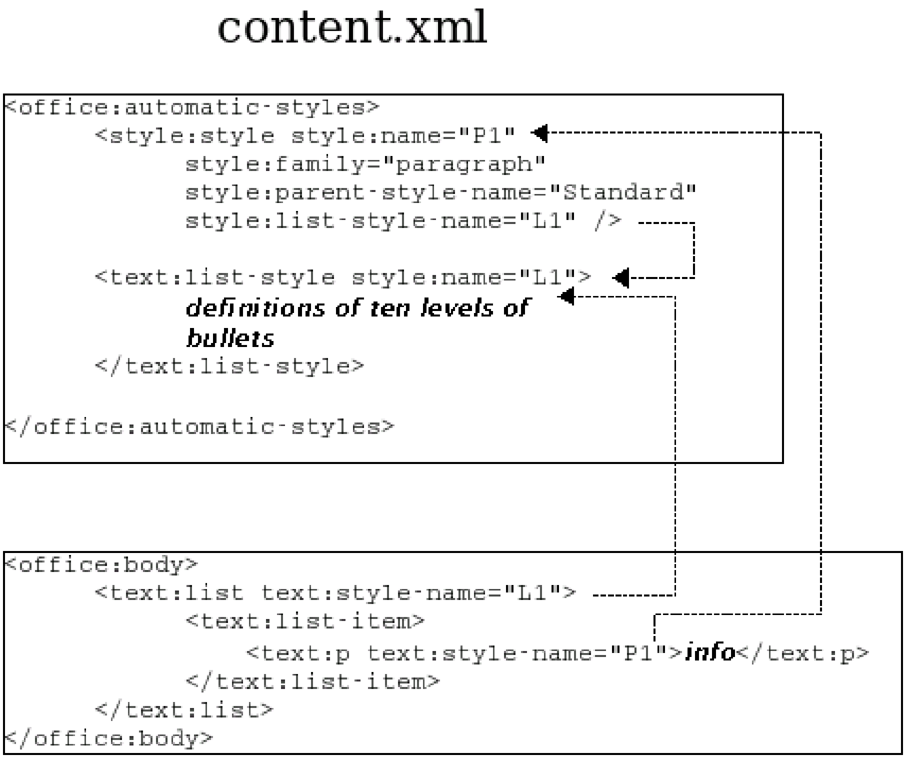
```
<office:automatic-styles>
    <style:style style:name="P1"
        style:family="paragraph"
        style:parent-style-name="Standard"
        style:master-page-name="Standard">
        <style:properties style:page-number="0" />
    </style:style>
    <style:style style:name="P2" style:family="paragraph"
        style:parent-style-name="Text body">
        <style:paragraph-properties fo:break-before="page" />
    </style:style>
    <!-- etc -->
</office:automatic-styles>

<office:body>
    <office:text>
        <text:p text:style-name="P1">Page one.</text:p>
        <text:p text:style-name="P2">Page two.</text:p>
    </office:text>
</office:body>
```

Bulleted, Numbered, and Outlined Lists

The specifications for bulleted, numbered, and outline lists are contained entirely within the `content.xml` document, and are related to one another as shown in Figure 3.8, "Relationship Among Elements When Specifying Lists".

The essential information is contained in the `<text:list-style>` element, which contains ten `<text:list-level-style-bullet>` elements if the list is all bulleted, `<text:list-level-style-number>` elements if the list is all numbered, or a mixture if the list is outlined. In OpenOffice.org, there are ten of these elements because the application allows a maximum of ten list levels.

Figure 3.8. Relationship Among Elements When Specifying Lists



```
<office:automatic-styles>
      <style:style style:name="P1"
            style:family="paragraph"
            style:parent-style-name="Standard"
            style:list-style-name="L1" />

      <text:list-style style:name="L1">
            definitions of ten levels of
            bullets
      </text:list-style>

</office:automatic-styles>
```

```
<office:body>
      <text:list text:style-name="L1">
            <text:list-item>
                  <text:p text:style-name="P1">info</text:p>
            </text:list-item>
      </text:list>
</office:body>
```

The `<text:list-level-style-bullet>` and `<text:list-level-style-number>` elements have the following attributes in common:

`text:level`
: An integer in the range 1 to 10.

`text:style-name`
: The value is a reference to a style in the `styles.xml` file. For bullets, OpenOffice.org sets this to `"Bullet_20_Symbols"`, which is the name of a `<style:style>` that defines the font to be used for the bullet symbols. For numbering, the value is `"Numbering_20_Symbols"`, which is the name of an empty `<style:style>` element.

`text:bullet-char`
: A single Unicode character to be used as the bullet; used for bulleted levels only.

The following attributes apply only to numbered lists.

`style:num-format`
: The format of the number; possible values are `1` for arabic numerals, `a` and `A` for lowercase and uppercase lettering, and `i` and `I` for lowercase and uppercase roman numerals.

`style:num-suffix`
: The character or characters to place after the number. OpenOffice.org creates this optional attribute for bulleted lists, even though it has no purpose.

`style:num-prefix`
: The character or characters to place before the number.

Thus, if you had numbered items of the form `(a)`, `(b)`, etc., the appropriate attributes would be `style:num-prefix="("`, `style:num-format="a"`, and `style:num-suffix=")"`.

Each `<text:list-level-style-number>` or `<text:list-level-style-bullet>` element contains a `<style:list-level-properties>` which specifies:

`text:min-label-width`
: The minimum space to allocate to the number or bullet.

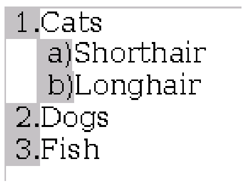
`text:space-before`
: The amount to indent before the bullet. This attribute does not appear for the first level of bullet.

For bullets, the `<style:list-level-properties>` is followed by a `<style:text-properties>` that gives the `style:font-name` of the bullet symbol.

Once you have established the list styles, you use them by creating a `<text:list>`. This element will have a `text:style-name` attribute that refers to the list style you want, and that style determines whether the list is bulleted or numbered. Each item in the list will be contained within a `<text:list-item>` element.

Example 3.17, "XML for an Outline List" shows the relevant part of the XML that produces the outline shown in Figure 3.9, "Screenshot of an Outline List".

Figure 3.9. Screenshot of an Outline List



1.Cats
a)Shorthair
b)Longhair
2.Dogs
3.Fish

Example 3.17. XML for an Outline List

```
<office:automatic-styles>

    <text:list-style style:name="L1">
        <text:list-level-style-number text:level="1"
            text:style-name="Numbering_20_Symbols"
            style:num-prefix=" " style:num-suffix="."
            style:num-format="1">
            <style:list-level-properties
                text:space-before="0.25in"
                text:min-label-width="0.25in"/>
        </text:list-level-style-number>
        <text:list-level-style-number text:level="2"
            text:style-name="Numbering_20_Symbols"
            style:num-prefix=" " style:num-suffix=")"
            style:num-format="a">
            <style:list-level-properties
                text:space-before="0.5in"
                text:min-label-width="0.25in"/>
        </text:list-level-style-number>
        <text:list-level-style-bullet text:level="3"
            text:style-name="Bullet_20_Symbols"
            style:num-prefix=" " style:num-suffix=" "
            text:bullet-char="•">
            <style:list-level-properties
                text:space-before="0.75in"
                text:min-label-width="0.25in"/>
            <style:text-properties style:font-name="StarSymbol"/>
        </text:list-level-style-bullet>
        <!-- the bullet is repeated for levels 4 through 10 -->
    </text:list-style>

    <style:style style:name="P1"
        style:family="paragraph"
        style:parent-style-name="Standard"
        style:list-style-name="L1"/>
</office:automatic-styles>

<office:body>
    <office:text>
        <text:list text:style-name="L1">
            <text:list-item>
                <text:p text:style-name="P1">Cats</text:p>
                <text:list>
                    <text:list-item>
                        <text:p ►
text:style-name="P1">Shorthair</text:p>
                    </text:list-item>
                    <text:list-item>
                        <text:p text:style-name="P1">Longhair</text:p>
                    </text:list-item>
                </text:list>
            </text:list-item>

            <text:list-item>
                <text:p text:style-name="P1">Dogs</text:p>
            </text:list-item>

            <text:list-item>
```

```
                <text:p text:style-name="P1">Fish</text:p>
            </text:list-item>
        </text:list>
    </office:text>
</office:body>
```

Case Study: Adding Headings to a Document

In Example 3.10, “Extracting Headings from an OpenDocument File”, we extracted the headings from an OpenOffice.org document and placed them into a new document. In this case study, we will add the headings to the current document. They will be represented as a bulleted list in a section at the beginning of the document. This is definitely the most ambitious example so far. In fact, at several points we almost abandoned the idea in favor of a simpler example. However, we realized that we would have to handle the tricky details at some point, and there was no time like the present.

This program is written in Java and is run from the command line. It takes two arguments: the name of the original file, and the name of the new file. We had considered simply modifying the original file in place, but if you ran the program twice you’d end up with two sets of bullet items. Here’s the plan:

- Copy all the JAR file entries other than `content.xml` directly to the new file.
- Parse the `content.xml` JAR entry and build a document tree.
- Add a new paragraph style, list style, and section style to the document tree. This avoids conflicts with existing styles. It also requires us to find the largest paragraph, list, and section style numbers so we can assign a unique identifier to our new styles.[4]
- Add a new section at beginning of the document body.
- Add a bulleted list to the section. The bullet levels correspond to the heading levels. This is an interesting process in itself, and we’ll talk about it when we get there.
- Write the updated document tree to the output file as the new `content.xml` JAR entry.

We start with the declarations of imported classes, which is not terribly illuminating, so we won’t duplicate it here. You can see them in the `AddOutline.java` file in the `ch03` directory in the downloadable example files.

We continue with the class declaration and class variables. In addition to finding the last list, paragraph, and section style numbers, we also need to keep track of their location in the document tree so that we can add our new styles immediately after the existing ones.

[4] The style names are in the format `P`, `L`, and `Sect` followed by an integer; this is the “number” we are referring to.

```
public class AddOutline
{
    /* The parsed document */
    protected Document document = null;

    /* Permanent pointer to root element of output document */
    protected Element documentRoot;

    /* List of heading elements */
    protected NodeList headingElements;

    /* Last numbers for paragraph, list, and section styles */
    int         lastParaNumber = 0;
    int         lastListNumber = 0;
    int         lastSectionNumber = 0;

    /* Node locations of last paragraph, list, and section styles */
    Node        lastParaNode = null;
    Node        lastListNode = null;
    Node        lastSectionNode = null;

    /* File descriptors */
    File        inFile;
    File        outFile;

    /* Streams for reading and writing jar files */
    JarInputStream  inJar;
    JarOutputStream outJar;
```

The main program is simplicity itself; it checks for the proper number of arguments, creates a class, and hands the arguments to the `modifyDocument` method.

```
public static void main(String argv[]) {

    // check for proper number of arguments
    if (argv.length != 2) {
        System.err.println("usage: java AddOutline filename
newfilename");
        System.exit(1);
    }

    AddOutline adder = null;

    adder = new AddOutline();
    adder.modifyDocument(argv);

} // main(String[])
```

Here is the `modifyDocument` method

```
protected void modifyDocument(String argv[])
{
    JarEntry inEntry;
    JarEntry outEntry;

    /* Create file descriptors */
    inFile = new File(argv[0]); ❶
    outFile = new File(argv[1]);
```

```
openInputFile();

/* Get the manifest from the input file */
Manifest manifest = inJar.getManifest( );

/* Open output file, copying manifest if it existed */
try
{
    if (manifest == null)
    {
        outJar = new JarOutputStream(new
           FileOutputStream(outFile));
    }
    else
    {
        outJar = new JarOutputStream(new
            FileOutputStream(outFile),
            manifest);
    }
}
catch (IOException e)
{
    System.err.println("Unable to open output file.");
    System.exit(1);
}

try ❷
{
    byte    buffer[] = new byte[16384];
    int     nRead;

    while ((inEntry = inJar.getNextJarEntry()) != null)
    {
        if (!inEntry.getName().equals("content.xml"))
        {
            /*
             * Create output entry based on information in
             * corresponding input entry
             */
            outEntry = new JarEntry(inEntry);
            outJar.putNextEntry(outEntry);

            /* Copy data */
            while ((nRead = inJar.read(buffer, 0, 16384)) != -1)
            {
                outJar.write(buffer, 0, nRead);
            }
        }
    }

    inJar.close(); ❸
    openInputFile();

    while ((inEntry = inJar.getNextJarEntry()) != null &&
        !(inEntry.getName().equals("content.xml")))
    {
        /* do nothing */
    }
```

```
        /*
         * Create output entry based on information in
         * corresponding input entry, but update its
         * timestamp.
         */
        outEntry = new JarEntry(inEntry);
        outEntry.setTime(new Date().getTime());
        outJar.putNextEntry(outEntry);

        ❹
        document = readContent();   /* parse content.xml */
        processContent();   /* add styles and bulleted list */
        writeContent();     /* write it to output JAR file */

        outJar.close();
    }
    catch (IOException e)
    {
        System.err.println("Error while creating new file");
        e.printStackTrace();
    }
}
```

❶ Start by opening the input file, and copying its manifest file (if any) to the output file. We use a method to open the input file, because we'll need to do it twice.

❷ The next stage is to copy all the JAR entries other than the `content.xml` file. The `try` block that starts here extends to nearly the end of the method; any error gives a generic error message and a stack trace.

❸ We must then close the input file and re-open it in order to find the `content.xml` JAR entry and process it. You may be wondering why we didn't just do this as one loop, copying all the entries except `content.xml` and processing it specially. We tried that, and it doesn't work; the XML parser closes its input file when it finishes, so the loop would fail with an "Input stream closed" error when it got to the entry after `content.xml`

In this second loop, we use a `while` loop with no statements in the loop body to get to the desired entry in the JAR file.

❹ Here's where the main work of creating the outline occurs; we will look at it in detail shortly.

Here's the code that opens the input file; nothing special to see here—keep moving along.

```
public void openInputFile()
{
    try
    {
        inJar = new JarInputStream(new FileInputStream(inFile));
    }
    catch (IOException e)
    {
        System.err.println("Unable to open input file.");
        System.exit(1);
    }
}
```

It doesn’t take much code to parse the XML file either. We need to create a parser, set its input source to the entry from the JAR file, and get the result when everything finishes. We must also have Xerces ignore any DTDs it might find in the `<!DOCTYPE>`; the relevant line is shown in boldface, and uses the `ResolveDTD` class described in the section called “Getting Rid of the DTD”.

```
public Document readContent( )
{
    try
    {
        DOMParser parser = new org.apache.xerces.parsers.DOMParser();
        parser.setEntityResolver(new ResolveDTD());
        parser.parse(new InputSource(inJar));
        return parser.getDocument();
    }
    catch (Exception e)
    {
        e.printStackTrace(System.err);
        return null;
    }
}
```

Now we must add the styles and text to the document tree; this is done in method `processContent`. The lines shown in boldface are methods that do much of the heavy lifting; we will look at each of those methods individually.

```
public void processContent()
{
    Node        autoStyles;
    Element     officeBodyStart;    /* the <office:body> element */
    Element     officeTextStart;    /* the <office:text> element */
    Element     textStart;          /* place to insert new text */
    Element     element;

    if (document == null)

    {
        return;
    }

    documentRoot = (Element) document.getDocumentElement();

    headingElements = document.getElementsByTagName("text:h"); ❶
    if (headingElements.getLength() == 0)
    {
        return;
    }

    autoStyles = findFirstChild(documentRoot,
        "office:automatic-styles"); ❷
    findLastItems(autoStyles);
    /*
     * Prepare to add the new styles by going to the next
     * available number. We'll insert the new style before
     * the next sibling of the last node. We have to do this
     * crazy thing because there is no "insertAfter" method.   ❸
     */
    lastParaNumber++;
    lastListNumber++;
    lastSectionNumber++;
    if (lastParaNode != null)
    {
        lastParaNode = lastParaNode.getNextSibling();
    }
    if (lastListNode != null)
    {
        lastListNode = lastListNode.getNextSibling();
    }
    if (lastSectionNode != null)
    {
        lastSectionNode = lastSectionNode.getNextSibling();
    }

    /*
     * Create a <style:style> element for the new paragraph,
     * set its attributes and insert it after the last paragraph
     * style.
     */
    element = document.createElement("style:style");
    element.setAttribute("style:name", "P" + lastParaNumber);
    element.setAttribute("style:family", "paragraph");
    element.setAttribute("style:list-style-name", "L" +
lastListNumber);
    element.setAttribute("style:parent-style-name", "Standard");
    autoStyles.insertBefore(element, lastParaNode);

    /*
     * Create a <style:style> element for the new section,
     * set its attributes and insert it after the last section
     * style.
     */
    element = document.createElement("style:style");
    element.setAttribute("style:name", "Sect" + lastSectionNumber);
    element.setAttribute("style:family", "section");
    addSectionProperties(element);
    autoStyles.insertBefore(element, lastSectionNode);

    /*
     * Create a <text:list-style> element for the new list,
     * set its attributes and insert it after the last list
     * style.
     */
    element = document.createElement("text:list-style");
    element.setAttribute("style:name", "L" + lastParaNumber);
    addBullets(element);

    autoStyles.insertBefore(element, lastListNode);

    /*
     * Now proceed to where we will add text;
     * it's just after the first <text:sequence-decls>
     * in the <office:text> in the <office:body>
     */
    officeBodyStart = findFirstChild(documentRoot, "office:body");
    officeTextStart = findFirstChild(officeBodyStart, "office:text");
    textStart = findFirstChild(officeTextStart,
       "text:sequence-decls");
    textStart = getNextElementSibling( textStart ); ❹

    /*
     * Add a section
     */
    element = document.createElement("text:section");
    element.setAttribute("text:style-name",
       "Sect" + lastSectionNumber);
    element.setAttribute("text:name", "Section" + lastSectionNumber);
    addHeadings(element);
    officeTextStart.insertBefore(element, textStart);
}
```

❶ Gather up all the `<text:h>` elements in the document; if there are none, then our job here is done.

❷ This line uses a utility method (`findFirstChild`) to find the first child of the document root whose element name is `office:automatic-styles`.

❸ Expanding on the comment: we want to place the new paragraph style after the last paragraph style in the current document, the new section style after the last existing style, etc. However, there's no `insertAfter` method, so we have to `insertBefore` the next sibling of the desired node.

❹ We haven't discussed the `<text:sequence-decls>` element yet; it's used for numbering items in OpenOffice.org documents. The main text in your document normally immediately follows this element. However, if you have any text nodes (such as newlines) between elements, the DOM `getNextSibling` will not give us what we want; thus, we use our own utility method `getNextElementSibling`.

As long as we're talking about the utility routines, they're fairly short, so we may as well present them here:

```
/*
 * Find first element with a given tag name
 * among the children of the given node.
 */
public Element findFirstChild(Node startNode, String tagName)
{
    startNode = startNode.getFirstChild();
    while (! (startNode != null &&
        startNode.getNodeType() == Node.ELEMENT_NODE &&
        ((Element)startNode).getTagName().equals(tagName)))
    {
        startNode = startNode.getNextSibling();
    }
    return (Element) startNode;
}

/*
```

```
 *  Find next sibling that is an element
 */
public Element getNextElementSibling( Node node )
{
    node = node.getNextSibling();
    while (node != null &&
        node.getNodeType() != Node.ELEMENT_NODE)
    {
        node = node.getNextSibling();
    }
    return (Element) node;
}
```

The next method, `addSectionProperties` is more of a convenience method than anything else:

```
/*
 * Add the appropriate properties to make a single-column
 * section
 */
public void addSectionProperties(Element sectionStyle)
{
    Element properties;
    Element columns;

    properties = document.createElement("style:section-properties");
    properties.setAttribute("text:dont-balance-text-columns",
        "false");

    columns = document.createElement("style:columns");
    columns.setAttribute("fo:column-count", "0");
    columns.setAttribute("fo:column-gap", "0cm");

    properties.appendChild(columns);
    sectionStyle.appendChild(properties);
}
```

The `addBullets` method is also quite straightforward; it’s a simple loop to create the ten levels of bullet styles. All levels except the first have `text:space-before`. Ordinarily OpenOffice.org creates its bullet styles with the StarSymbol font, and references a style named `Bullet Symbols` in the `styles.xml` file. We are dispensing with that, so our new document will use the bullet symbol from the default font.

```
/*
 * Add the ten bullet styles to the <text:list-style> element.
 */
public void addBullets(Element listLevelStyle)
{
    int     level;
    Element bullet;
    Element properties;
    for (level = 1; level <= 10; level++)
    {
        bullet =
            document.createElement("text:list-level-style-bullet");
        bullet.setAttribute("text:level", Integer.toString(level));
```

```
            bullet.setAttribute("text:bullet-char", "\u2022");

            properties = document.createElement(
                "style:list-level-properties");
            if (level != 1)
            {
                properties.setAttribute("text:space-before",
                    Double.toString((level-1) * 0.5) + "cm");
            }
            properties.setAttribute("text:min-label-width", "0.5cm");
            bullet.appendChild(properties);
            listLevelStyle.appendChild(bullet);
        }
}
```

Adding the headings is conceptually a recursive process, since each new level of heading opens a nested list. However, there is no guarantee that heading levels will increase and decrease sequentially; a level three heading can be followed by a level seven heading, followed by a level one heading. (This is not good document design, but it is certainly possible.) Thus, rather than write this method recursively, we decided to use an array to simulate a stack. Here's the pseudo-code:

```
1. Set the current level to zero
2. For each heading in the document:
    a. If the heading level is greater than the current level, open one new
       <text:list> element for each level from the current level to
       the given heading level.
    b. If the heading level is less than the current level, close off open
       lists. Start at the current level and append that list to the end of the
       previous level. Then append *that* list to its predecessor, and so on
       until you get to the current level.
    c. Now accumulate all the text from the heading (skipping over
       formatting information and elements), and append it to the list at
       the current level.
3. If you are not at level zero, close off all currently open lists as in step 2b.
```

```
/* Add headings to a section */
public void addHeadings(Element startElement)
{
    int currentLevel = 0;
    int headingLevel;
    int i;
    int level;
    Element ulist[] = new Element[10];
    Element listItem;
    Element paragraph;
    Text    textNode;

    for (i=0; i < headingElements.getLength(); i++)
    {
        headingLevel = Integer.parseInt(
            ((Element)headingElements.item(i)).getAttribute(
                "text:level"));
        if (headingLevel > currentLevel)
        {
            for (level = currentLevel; level < headingLevel; level++)
            {
                ulist[level] = document.createElement(   ❶
                    "text:list");
                if (level == 0)
                {
                    ulist[level].setAttribute("text:style-name",
                        "L" + lastListNumber);
                }
            }
            currentLevel = headingLevel;
        }
        else if (headingLevel < currentLevel)
        {
            closeLists(ulist, currentLevel, headingLevel); ❷
            currentLevel = headingLevel;
        }

        /* Now append this heading as an item to current level */

        listItem = document.createElement("text:list-item");   ❸
        paragraph = document.createElement("text:p");
        paragraph.setAttribute( "text:style-name", "P"
            + lastParaNumber );
        textNode = document.createTextNode("");
        textNode = accumulateText(
            (Element) headingElements.item(i),
            textNode);
        paragraph.appendChild(textNode);
        listItem.appendChild(paragraph);
        ulist[currentLevel-1].appendChild(listItem);

    }
    if (currentLevel != 1) ❹
    {
        closeLists(ulist, currentLevel, 1);
    }
    startElement.appendChild(ulist[0]);
}
```

❶ We add levels by creating `<text:list>` elements. Only the first level unordered list has a `text:style-name` attribute.

❷ The work of closing lists when the level decreases has been passed on to a separate routine.

❸ No matter whether we have added levels, closed levels, or are at the same level, we have to add a `<text:list-item>` at the current level. The `accumulateText` method gathers all the text nodes in the heading.

❹ Another call to the `closeLists` method closes any nested lists.

Here is the `closeLists` method. Each `<text:list>` that is being closed is within a `<text:list-item>` element of its parent list.

```
/*
 * Join elements in the ulist[] array to close all open lists
 * from currentLevel back down to newLevel
 */
public void closeLists(Element ulist[], int currentLevel, int
newLevel)
{
    int i;
    Element listItem;
    for (i = currentLevel-1; i > newLevel-1; i--)
    {
        listItem = document.createElement("text:list-item");
        listItem.appendChild(ulist[i]);
        ulist[i-1].appendChild(listItem);
    }
}
```

Finally, the `accumulateText` method, which recursively visits all the child nodes of the `<text:h>` element in question. Tabs and line breaks are replaced with a single blank each; any other elements are ignored.

```
/*
 * Return a Text node that contains all the accumulated text
 * from the child nodes of the given element.
 */
public Text accumulateText(Element element, Text text)
{
    Node node = element.getFirstChild();
    while (node != null)
    {
        if (node.getNodeType() == Node.TEXT_NODE)
        {
            text.appendData(((Text) node).getData());
        }
        else if (node.getNodeType() == Node.ELEMENT_NODE)
        {
            if (((Element) node).getTagName().equals(
                "text:tab-stop") ||
                ((Element) node).getTagName().equals(
                "text:line-break"))
            {
                text.appendData(" ");
            }
            else

            {
                text = accumulateText((Element) node, text);
            }
        }
        node = node.getNextSibling();
    }
    return text;
}
```

This covers all the methods used to process the document tree. All that remains is the method to serialize the document tree, writing it to the new document's `content.xml` JAR file entry.

```
public void writeContent()
{
    if (document == null)
    {
        return;
    }
    PrintWriter out = null;
    try
    {
        out =
        new PrintWriter(new OutputStreamWriter(outJar, "UTF-8"));  ❶
    }
    catch (Exception e)
    {
        System.out.println("Error creating output stream");
        System.out.println(e.getMessage());
        System.exit(1);
    }
    OutputFormat oFormat = new OutputFormat("xml", "UTF-8", false); ❷
    XMLSerializer serial = new XMLSerializer(out, oFormat); ❸
    try
    {
        serial.serialize(document);
    }
    catch (java.io.IOException e)
    {
        System.out.println(e.getMessage());
    }
}
```

❶ First, construct an output stream with your favorite encoding method

❷ A serializer requires an output format. This constructor's three parameters are the output method (which is normally one of `"xml"`, `"html"`, or `"text"`); the character encoding, which should be `"UTF-8"` to keep your international clients happy; and a boolean that tells whether the output should be indented or not. Set this to `false` to avoid unwanted whitespace text nodes.

❸ The `OutputFormat` is used when creating the serializer.

Chapter 4. Text Documents—Advanced

Frames

A frame in an OpenDocument word processing document is much like a section; it's an independent area of text which may have multiple columns. The difference between a frame and a section is that a frame may "float" and have the main text wrap around it. Frames are also *anchored* to the page, a pargraph, or an individual character. They may also act as though they are just another character in the stream of the text.

Style Information for Frames

Each frame will have a `<style:style>` element whose `style:name` begins with `fr` and whose `style:family` is `graphic` (yes, frames are actually considered to be graphic objects). Its `style:parent-style-name` will be `Frame`.

Within the `<style:style>` is a `<style:graphic-properties>` element with these relevant attributes:

`style:vertical-rel`
: This attribute tells where the frame is anchored: `page`, `paragraph-content`, or `char`. If the frame is anchored as a character, then this attribute has the value `baseline`.

`style:vertical-pos`
: This gives the vertical position with respect to the anchor: `top`, `middle`, or `bottom`. If you have manually adjusted a frame by moving it, then this value will be `from-top`, and the offset will be in the body of the document.

`style:horizontal-rel`
: Depending upon the anchorage of the frame, this attribute can have the following values: `page` and `page-content` (the entire page or just the text area), `page-start-margin` and `page-end-margin`, `paragraph` and `paragraph-content`, `paragraph-start-margin` and `paragraph-end-margin`, or `char`. If you have frames nested in frames, you may use the following values as well: `frame` and `frame-content` (the entire frame or just the occupied area), `frame-start-margin`, and `frame-end-margin`.

`style:horizontal-pos`
: The values used for this attribute are `left`, `center`, `right`. If you have manually adjusted a frame by moving it, then this value will be `from-left`, and the offset will be in the body of the document.

`style:wrap`
: How should text wrap around this frame? `none`, `left` (all text appears at the left of the frame), `right` (all text appears to the right of the frame), `parallel` (text appears on both sides of the frame), `dynamic` (what OpenOffice.org calls “optimal” page wrap), and `run-through` (the text appears behind the frame). [5]

`style:number-wrapped-paragraphs`
: This attribute has the value `no-limit` unless you have checked the “First paragraph” option, in which case this attribute is not present.

A frame can have borders and columns. The borders are set as described in the section called “Borders and Padding”, and the columns are set as described in the section called “Sections”. A frame’s background color is set with the `fo:background-color` attribute; the value is a six-digit hex value. You may set the `style:background-transparency` attribute to a value from `0%` to `100%`.

Body Information for Frames

In the body of the document, each frame is represented by a `<draw:text-box>` element, with these attributes:

`draw:style-name`
: A reference to the corresponding `<style:style>` for the frame within the `<office:automatic-styles>` section.

`draw:name`
: OpenOffice.org will assign a value of the form `Frame`*n* where *n* is an integer; within OpenOffice.org, this appears within the Format … Frame … Options dialog pane.

`text:anchor-type`
: This attribute has one of the values `page`, `frame`, `paragraph`, `char`, or `as-char`.

Note

A frame that is anchored to the page appears between the `<text:sequence-decls>` element and the beginning of the page text, and has a `text:anchor-page-number` to identify the page to which it is anchored.

`svg:width`
: The width of the frame.

[5] If you want the frame in the background, then set the `style:run-through` attribute to `background` instead of `foreground`.

`svg:x`, `svg:y`

These are included only if the frame is not left, right, or center aligned.

`draw:z-index`

This attribute is used to determine the placement of overlapping elements; the larger the number, the closer to the “front” it is.

Inside the `<draw:frame>` element is a `<draw:text-box>` which has a `fo:min-height` attribute giving the frame’s minimum height. There are optional `fo:max-height` and `fo:max-width` attributes as well, which may be expressed as a *length* or a percentage. The actual text for the frame is inside the `<draw:text-box>` element.

Example 4.1, “XML Representation of a Frame” shows the style and body information for a frame that has text that wraps on the left.

Example 4.1. XML Representation of a Frame

```
<!-- in the style section -->
<style:style style:name="fr4" style:family="graphic"
  style:parent-style-name="Frame">
    <style:graphic-properties style:run-through="foreground"
      style:wrap="left"
      style:number-wrapped-paragraphs="no-limit"
      style:vertical-pos="top" style:vertical-rel="paragraph"
      style:horizontal-pos="center"
      style:horizontal-rel="paragraph" />
</style:style>

<!-- in the body section -->
<draw:frame draw:style-name="fr4" draw:name="Frame3"
  text:anchor-type="paragraph" svg:width="3cm"
  draw:z-index="2">
    <draw:text-box fo:min-height="0.2in">
        <text:p text:style-name="P1">
            A centered frame with text wrapping on its left.
        </text:p>
    </draw:text-box>
</draw:frame>
```

Inserting Images in Text

When you insert an image into an OpenDocument file, the application will store a copy of that image file in the `Pictures` directory and assign it an internal filename that, in the case of OpenOffice.org, looks something like this: `100000000000001800000018374E562F.png`. The filename extension corresponds to the type of the original graphic.

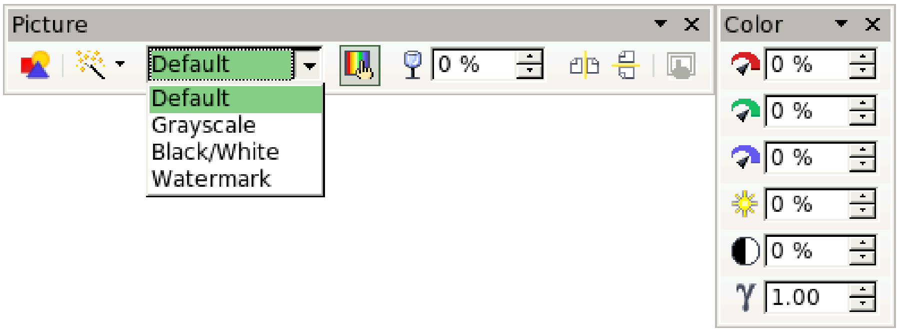
As with many other items in OpenDocument, the specification for an image is contained both within the `<office:automatic-styles>` and `<office:body>` elements.

Style Information for Images in Text

Each `<style:style>` element for an image has a `style:family` of `graphic` and a `style:parent-style-name` of `Graphics`. There is also a `style:name` attribute, which gives the name of this image syle.

The `<style:style>` element will contain a `<style:graphic-properties>` element that gives further information about the frame, exactly as described in the section called "Style Information for Frames". Additionally, you will find the following attributes in the `<style:properties>` element which correspond to the settings in the graphic object toolbars shown in Figure 4.1, "Graphic Object Toolbars".

Figure 4.1. Graphic Object Toolbars



`draw:color-mode`
: This attribute has values which correspond to the drop-down menu: `standard`, `greyscale`, `mono`, and `watermark`.

`draw:transparency`
: Ranges from `0%` (the default, opaque) to `100%` (totally transparent).

`style:mirror`
: This attribute has the value `horizontal`, `vertical`, or `horizontal vertical` depending upon which of the icons on the toolbar have been selected.

`fo:clip`
: If you have chosen to clip the image, then the clip rectangle is specified as this example: `rect(0.2cm 0.25cm 0cm 0.05cm)` The values specify the amount of clipping at the top, right, bottom and left of the image. Hint: the values go clockwise from 12:00. (This option is not in the toolbar; it's in the graphics dialog box.)

`draw:red`, `draw:green`, `draw:blue`
: The amount of extra red, green, or blue for the image. The value ranges from `-100%` (none of the color) to `100%` (full amount of the color). The default is `0%`.

`draw:luminance`
: The extra brightness of the images; range goes from `-100%` to `100%`. The default is `0%`.

`draw:contrast`
: Amount of extra contrast, also ranging from `-100%` to `100%`. The default is `0%`.

`draw:gamma`
: This value adjusts the brightness of midtone levels (rather than luminance, which brightens everything). Values range from `10%` upwards. The default is `100%`

Body Information for Images in Text

Each image in the text is represented by a `<draw:image>` contained in a `<draw:frame>` element. The `<draw:image>` element has these attributes:

`draw:style-name`
: A reference to a `<style:style>` within the `<office:automatic-styles>` section. The name begins with the letters `fr`, since images are represented as frames, as discussed in the section called "Frames".

`draw:name`
: This is an optional attribute; the Openoffice.org application will assign a value of the form `Graphics`*`n`* where *n* is an integer.

`xlink:href`
: A reference to the image file; it begins with `Pictures/` and is followed by the internal file name.

`svg:width`, `svg:height`
: This is the size of your image, with scaling factors taken into account. For example, if your original picture is one centimeter by one centimeter and you scale it to 75% horizontally and 125% vertically, the width will be `0.75cm` and the height `1.25cm`.

`svg:rel-width`, `svg:rel-height`
: These are percentages of the image's original size. If the value is `scale`, that means that the given dimension is kept in scale with the other dimension.

`xlink:type`, `xlink:show`, `xlink:actuate`
: These three items always have the values `simple` (for `xlink:type`), `embed` (for `xlink:show`), and `onLoad` (for `xlink:actuate`).

Background Images

A background image for a frame is entirely described in the style portion of your document. You need to put a `<style:background-image>` element within the frame's `<style:graphic-properties>` element. This element will have `xlink:href`, `xlink:actuate`, and `xlink:type` attributes as described in the section called "Body Information for Images in Text" (even though the attributes are in the style section, not the body section).

The `<style:background-image>` has the following additional properties:

`style:repeat`
: The background image can be tiled `repeat` (the default if you don't provide this attribute), stretched to fit the frame `stretch`, or appear at its normal size `no-repeat`.

`style:position`
: If the background image is not repeated, then you should tell where to place the image within the frame. The value of this attribute consists of two whitespace-separated values giving the vertical position of the image (`top`, `bottom`, or `center`) and horizontal position of the image (`left`, `right`, or `center`). The default is centered horizontally and vertically if you leave out this attribute.

`draw:opacity`
: The opacity of the background image, with values ranging from `0%` (transparent) to `100%` (opaque). Note that transparency equals 100% minus opacity.

Fields

OpenDocument allows you to enter fields with dynamic values into a document. These include date, time, and document information.

Date and Time Fields

To insert a date, use a `<text:date>` element with a `text:date-value` attribute, which contains the date in ISO8601 format. The `style:data-style-name` attribute is a reference to an automatic style that describes how to display the date. You may use `<text:date>` as an empty element; when OpenOffice.org saves a file, the element is given content equivalent to the date as your style would display it. This is probably done so that an application that ignores all elements and uses content only will produce a reasonable result. Nonetheless, OpenOffice.org uses only the value of the `text:date-value` when it displays the field.

To insert a time use a `<text:time>` element with a `text:time-value` attribute, which contains the time in ISO8601 format. The `style:data-style-name` attribute is a reference to an automatic style that describes how to display the time. As with `<text:date>`, you may use `<text:time>` as an empty element even though a document saved by an application such as OpenOffice.org will have content.

In both cases, these elements will display the current date and time; if you want them to display the date and time at which they were inserted into the document, add the attribute `text:fixed` with a value of `true`.

Page Numbering

The current page number is specified with the `<text:page-number>` element; its `text:select-page` attribute may have a value of `current`, `previous`, or `next`. Your best choice is clearly `current`.

The page count is specified with the `<text:page-count>` element. Again, OpenOffice.org puts content in these elements, but you don’t have to.

Document Information

The subject, title, and author fields get their information from elements in the `meta.xml` file, as shown in Table 4.1, “Document Information Fields”. You may create these as empty elements; the application should fill them with their current values.

Table 4.1. Document Information Fields

Text Document Element	`meta.xml` Element
`<text:subject>`	`<dc:subject>`
`<text:title>`	`<dc:title>`
`<text:initial-creator>`	`<meta:initial-creator>`

Footnotes and Endnotes

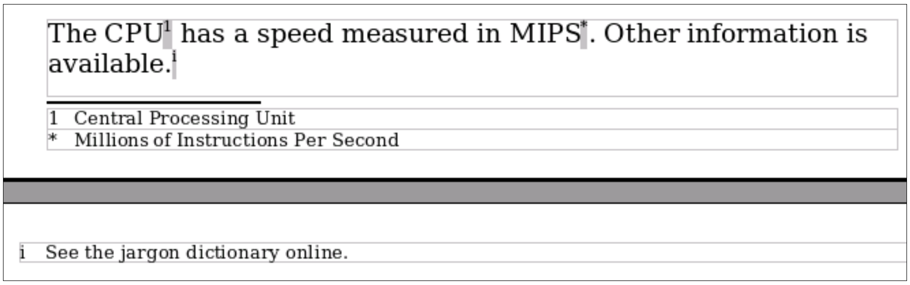
In the section called “Specifying a Page Master” we discussed how you set up the page layout to include room for footnotes. Within the document body, each footnote is contained within a `<text:note>` element, which has a unique `text:id` attribute of the form `ftn`*n*, where *n* is an integer. Footnotes set the `text:note-class` attribute to `footnote`; endnotes set the value to `endnote`.

Within the `<text:note>` is a `<text:note-citation>` element that describes the foonote/endnote marker. If you choose automatic numbering for the footnote, then the element's content is the footnote number. If you choose a character for the footnote marker, then the `<text:note-citation>` element contains the marker character. The marker is duplicated in the `text:label` attribute.

The `<text:note-citation>` is followed by the `<text:note-body>` element, which contains the text in your footnote.

Figure 4.2, "Footnotes and Endnotes" shows a numbered footnote, a footnote marked with an asterisk, and a numbered endnote. The corresponding XML is in Example 4.2, "Footnote and Endnote XML". The `Footnote` and `Endnote` styles come from the `styles.xml` file.

Figure 4.2. Footnotes and Endnotes



The CPU[1] has a speed measured in MIPS[*]. Other information is available.[i]

1 Central Processing Unit
* Millions of Instructions Per Second

i See the jargon dictionary online.

Example 4.2. Footnote and Endnote XML

```
<text:p text:style-name="Standard">
    The CPU
    <text:note text:id="ftn1"
      text:note-class="footnote">
        <text:note-citation>1</text:note-citation>
        <text:note-body>
            <text:p text:style-name="Footnote">
                Central Processing Unit
            </text:p>
        </text:note-body>
    </text:note>
    has a speed measured in MIPS
    <text:note text:id="ftn2"
      text:note-class="footnote">
        <text:note-citation text:label="*">*</text:note-citation>
        <text:note-body>
            <text:p text:style-name="Footnote">
                Millions of Instructions Per Second
            </text:p>
        </text:note-body>
    </text:note>. Other information is available.
    <text:note text:id="ftn3" text:note-class="endnote">
```

```
        <text:note-citation>i</text:note-citation>
        <text:note-body>
            <text:p text:style-name="Endnote">
                See the jargon dictionary online.
            </text:p>
        </text:note-body>
    </text:note>
</text:p>
```

Tracking Changes

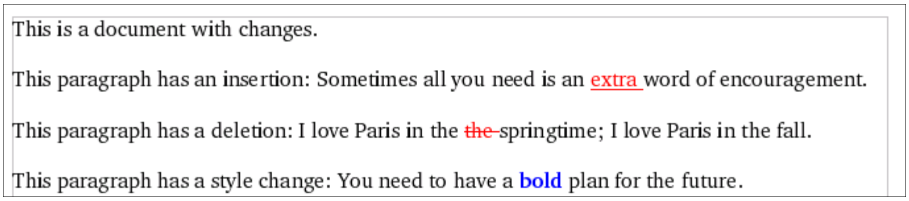
OpenDocument files track three types of changes: insertions, deletions, and format changes. These are all combined into a `<text:tracked-changes>` element at the beginning of the `<office:text>` element. Each change is contained in a `<text:changed-region>` element with a unique `text:id` attribute.

A `<text:changed-region>` contains one of three elements:

- `<text:insertion>`
- `<text:deletion>`
- `<text:format-change>`

These all share a common `<office:change-info>` element, which has `dc:creator` and `dc:date` attributes. In the case of a deletion, the `<office:change-info>` element is followed by the deleted material. (If it is only a single word, then it is enclosed in a copy of the parent `<text:p>` or `<text:h>` element from which it was deleted.) Figure 4.3, "Document with Changes Tracked" shows a section of a document with these three types of changes, and Example 4.3, "OpenDocument Change Tracking" shows the markup.

Figure 4.3. Document with Changes Tracked



This is a document with changes.

This paragraph has an insertion: Sometimes all you need is an extra word of encouragement.

This paragraph has a deletion: I love Paris in the ~~the~~ springtime; I love Paris in the fall.

This paragraph has a style change: You need to have a **bold** plan for the future.

Example 4.3. OpenDocument Change Tracking

```
<text:tracked-changes text:track-changes="false">
    <text:changed-region text:id="ct1256620744">
        <text:insertion>
            <office:change-info>
                <dc:creator>Phuong Nguyen</dc:creator>
                <dc:date>2005-06-04T20:57:00</dc:date>
            </office:change-info>
        </text:insertion>
    </text:changed-region>
    <text:changed-region text:id="ct1256247728">
        <text:deletion>
            <office:change-info>
                <dc:creator>Phuong Nguyen</dc:creator>
                <dc:date>2005-06-04T20:57:00</dc:date>
            </office:change-info>
            <text:p text:style-name="P1">the </text:p>
        </text:deletion>
    </text:changed-region>
    <text:changed-region text:id="ct1257148536">
        <text:format-change>
            <office:change-info>
                <dc:creator>Henry McCoy</dc:creator>
                <dc:date>2005-06-04T21:00:00</dc:date>
            </office:change-info>
        </text:format-change>
</text:changed-region>
```

In the body of the text, we must be able to determine where these accumulated changes have occurred. For deletions, a `<text:change>` element is placed where the deletion occurred; its `text:change-id` attribute will refer to the corresponding `text:id` of the `<text:changed-region>`.

For insertions and format changes, the start of the change is marked with an empty `<text:change-start>` element, and the end with an empty `<text:change-end>` element. Each of these also has a `text:change-id` attribute that refers to the corresponding `text:id`. Example 4.4, “Change Tracking in the Modified Text” shows the markup for the changes described in the preceding example.

Example 4.4. Change Tracking in the Modified Text

```
<text:p text:style-name="P1">This paragraph has an insertion:
Sometimes all you need is an
<text:change-start text:change-id="ct1256620744"/>extra
<text:change-end text:change-id="ct1256620744"/>word of
encouragement.</text:p>

<text:p text:style-name="P1"/>

<text:p text:style-name="P1">This paragraph has a deletion: I love
Paris
in the <text:change text:change-id="ct1256247728"/>springtime;
I love Paris in the fall.</text:p>

<text:p text:style-name="P1"/>

<text:p text:style-name="P1">This paragraph has a style change:
You need to have a
<text:change-start text:change-id="ct1257148536"/><text:span
text:style-name="T3">bold</text:span><text:change-end
text:change-id="ct1257148536"/> plan for the future.</text:p>
```

Tables in Text Documents

Text tables in OpenDocument are, as with HTML tables, made up of rows, each of which contains cells. Again, the information for the table layout is in the `<office:automatic-styles>` section and the table data within the `<office:body>` section. In this section, when we refer to a *length*, we mean a number followed by a length unit; for example, `3.5cm`.

Text Table Style Information

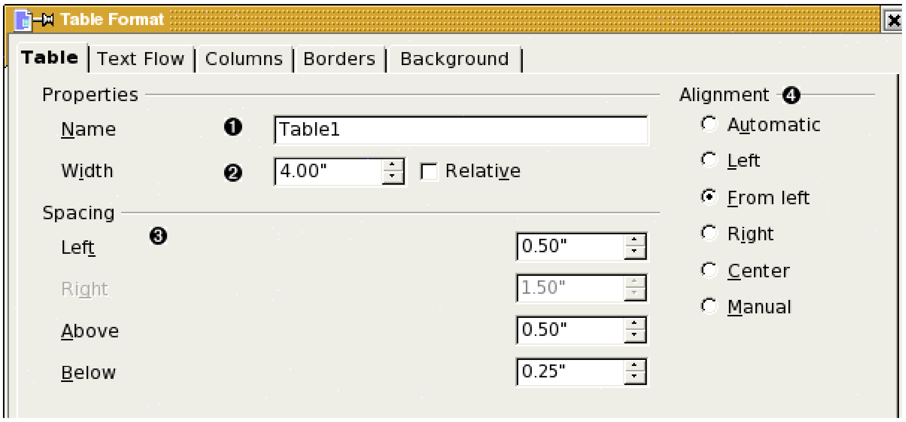
Within the `<office:automatic-styles>` element, you will find the following for each table:

- one `<style:style>` element with a `style:family` of `table` describing general table attributes.
- each unique column style has a `<style:style>` element with a `style:family` of `table-column` describing the column width.
- each unique row style has a `<style:style>` element with a `style:family` of `table-row` describing the row attributes.
- a `<style:style>` element with a `style:family` of `table-cell` for each different style of cell in the table.

Styling for the Entire Table

The information that styles the whole table corresponds to the information that is set in the portion of OpenOffice.org's "Format Table" dialog box shown in Figure 4.4, "Table Width and Spacing". In this section, we will be talking in terms of that particular application in order to make the concepts more concrete.

Figure 4.4. Table Width and Spacing



Properties

Name ❶ Table1

Width ❷ 4.00" ☐ Relative

Spacing ❸

Left 0.50"

Right 1.50"

Above 0.50"

Below 0.25"

Alignment ❹

○ Automatic
○ Left
● From left
○ Right
○ Center
○ Manual

The “whole table’s” `<style:style>` element has a `style:name` attribute containing the table name (1). Its child `<style:table-properties>` element contains the remaining information:

The `style:width` attribute (2) is a *length* that gives the total width of the table. If you check the “Relative” checkbox, then the width of the table as a percentage of the page width is stored in the `style:rel-width` attribute.

The spacing (3) is represented by `fo:margin-left`, `fo:margin-right`, `fo:margin-top`, and `fo:margin-bottom` attributes, which all have *length* values. The left and right margins plus the width always add up to the distance between the page margins.

In a document created by OpenOffice.org, the application sets some of the margins depending upon the setting of the alignment (4). The `table:align` attribute interacts with the margins in strange and wondrous ways when OpenOffice.org creates a document. If you are creating a document programmatically, set this attribute to

- `margins`, and set both left and right margins (the “Manual” option). If you don’t set them, then the table will fill the page width (the “Automatic” option).
- `left`, and set the left margin; OpenOffice.org sets the right margin. (This corresponds to the “Left” and “From left” entries in the dialog box.)
- `right`, and set the left margin; the right margin is set to zero. There is no “From right” option.
- `center`, and don’t set left or right margin, since OpenOffice.org will calculate them for you when the document is opened.

The top and bottom margins are up to you, no matter what alignment you choose.

Figure 4.5, “Other Table Properties” is the dialog box that corresponds to other attributes in the table’s `<style:table-properties>` element.

1. `break-after` and `break-before` (with values `page` or `column`)

2. If you don’t want tables to split across pages and columns, then set the `style:may-break-between-rows` attribute to `false`. The “allow row to break across pages and columns” is set in the row style specifications.

   “Keep with next paragraph” corresponds to the `fo:keep-with-next` attribute set to `always`.

3. “Repeat heading” is handled in the body of the text, not the style.

4. “Vertical alignment” is set on the cells that are currently highlighted, not on the entire table.

Figure 4.5. Other Table Properties

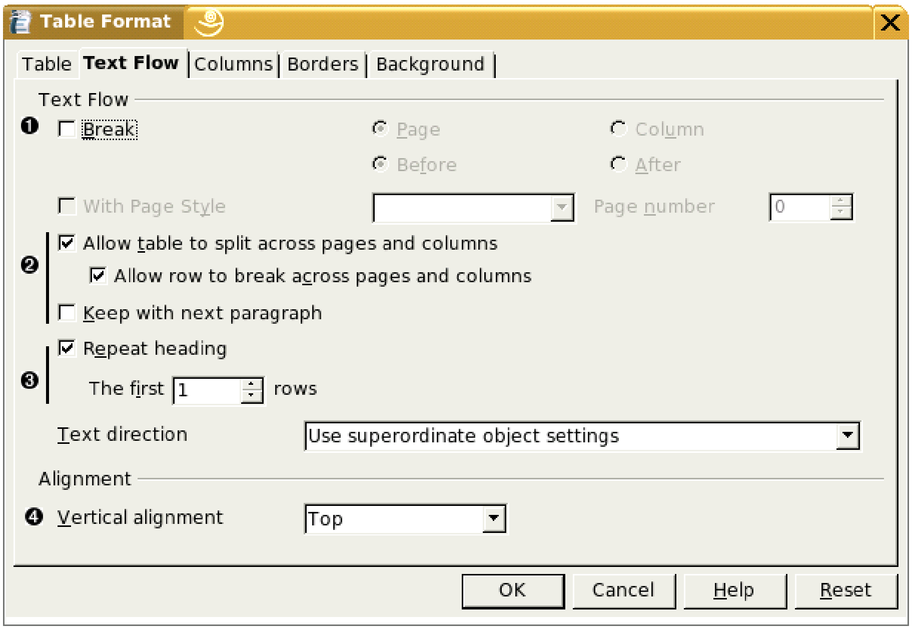


Table Format

Table | **Text Flow** | Columns | Borders | Background

Text Flow

❶ ☐ Break — ◉ Page ○ Column — ◉ Before ○ After

☐ With Page Style — Page number: 0

❷ ☑ Allow table to split across pages and columns

☑ Allow row to break across pages and columns

☐ Keep with next paragraph

❸ ☑ Repeat heading

The first 1 rows

Text direction: Use superordinate object settings

Alignment

❹ Vertical alignment: Top

OK | Cancel | Help | Reset

Styling for a Column

A column's `<style:style>` element has a a `style:family` of `table-column` and a `style:name` attribute of the form `demoTable.A` (table name followed by a column designator). If all the columns have the same width, then you will have only one such element.

A column's child `<style:table-column-properties>` element will have a `style:column-width` attribute, giving the column width as a *length* (`3.5cm`), and a `style:rel-column-width`, which is in the form of an integer followed by an asterisk. If you add up all the relative column widths, you will get a sum of 65535.

Styling for a Row

Rows have `<style:style>` elements with a `style:family` of `table-row`. Their `style:name` attribute has the form `demoTable.3` (table name, integer). This element contains a `<style:table-row-properties>` element that can specify such attributes as `style:row-height`, `style:min-row-height` (both *length* values), `style:use-optimal-row-height` (`true` or `false`), and `fo:keep-together`, which is set to `always` if you don't want to allow a row to break across a page or column boundary. (The default is `auto`.)

When creating an OpenDocument file, put in the row style specifications only if you don’t want the defaults.

Styling for Individual Cells

Cell styles are specified in `<style:style>` elements with a `style:family` of `table-cell`. Their `style:name` attribute has the form `demoTable.C3` (table name, column designator, and an integer). OpenOffice.org does not necessarily create a separate style for every cell; nor should you. Create only as many styles as are needed to acommodate all the different cell styles in your table.

The child `<style:table-cell-properties>` element is where you set

- `fo:vertical-align`, with values `top`, `middle`, or `bottom`.
- borders and padding, as described in the section called “Borders and Padding”. Make sure you don’t specify a border twice. For example, when OpenOffice.org creates a table, all the cells within a row have a left border, but only the last cell has a right border. (If you specified left and right borders on all cells, you’d see two lines between cells.) Similarly, within a column, only the top cell has a top border.
- background color, using the `fo:background-color` attribute.

In OpenOffice.org, these specifications are placed in the document with the table style first, followed by the column styles, followed by the row styles and cell styles for each different row and cell.

Text Table Body Information

Example 4.5, “Three by Two Table Without Repeating Headers” shows a three-by-two table without repeating headers.

Example 4.5. Three by Two Table Without Repeating Headers

```
<table:table table:name="demoTable" table:style-name="demoTable"> ❶
        <table:table-column table:style-name="demoTable.A"
          table:number-columns-repeated="3"/> ❷
        <table:table-row> ❸
                <table:table-cell table:style-name="demoTable.A1"  ❹
                  office:value-type="string">
                        <text:p>row 1, col 1</text:p>
                </table:table-cell>
                <table:table-cell table:style-name="demoTable.A1"
                  office:value-type="string">
                        <text:p>row 1, col 2</text:p>
                </table:table-cell>
                <table:table-cell table:style-name="demoTable.C1"
                  office:value-type="string">
                        <text:p>row 1, col 3</text:p>
                </table:table-cell>
    </table:table-row>
    <table:table-row>
                <table:table-cell table:style-name="demoTable.A2"
```

```
                  office:value-type="string">
                        <text:p>row 2, col 1</text:p>
                </table:table-cell>
                <table:table-cell table:style-name="demoTable.A2"
                  office:value-type="string">
                        <text:p>row 2, col 2</text:p>
                </table:table-cell>
                <table:table-cell table:style-name="demoTable.C2"
                  office:value-type="string">
                        <text:p>row 2, col 3</text:p>
                </table:table-cell>
        </table:table-row>
</table:table>
```

❶ A table's content is contained within a `<table:table>` element, which has a `table:style-name` attribute which references the `<style:style>` with the same name. It has non-alphanumeric characters translated to their hexadecimal equivalents. The `table:name` is the "display name" of the table that an application's user interface shows you.

❷ In this example, all three columns have the same style, so the `<table:table-column>` has a `table:number-columns-repeated` attribute. If all three columns had different styles, then the XML would contain three `<table:table-column>` elements, each with a different `table:style-name` reference.

❸ If you have specified that your table does not have repeating headings (item 3 in Figure 4.5, "Other Table Properties"), then the `<table:table-row>` elements follow immediately. If your table has repeating headers, then the first *n* rows of the table will be enclosed in a `<table:table-header-rows>` element, where *n* is the number of header rows you specified.

❹ Each `<table:table-row>` contains the `<table:table-cell>` elements for that row. Each `<table:table-cell>` has a `table:style-name` reference and a `office:value-type`, which has the value `string`. (We will encounter other values when we discuss spreadsheets.)

Merged Cells

Horizontally merged cells are simple in OpenDocument. The first of the cells gets a `table:number-columns-spanned` attribute, whose value is the number of columns that have been merged. That cell is followed by *n*-1 `<table:covered-table-cell>` elements. Thus, a cell that spans three columns might look like Example 4.6, "Cells Spanning Columns". In this example, the `text:style-name` attributes have been removed for ease of reading.

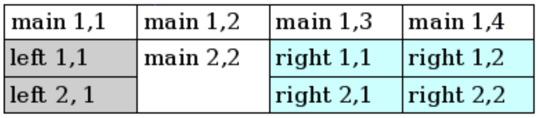
Example 4.6. Cells Spanning Columns

```
<table:table-cell table:style-name="horizspan.B2"
  table:number-columns-spanned="3" office:value-type="string">
    <text:p>This text spans three columns</text:p>
</table:table-cell>
<table:covered-table-cell />
<table:covered-table-cell />
```

Cells that span rows are an entirely different story. Rather than a simple `table:number-rows-spanned` attribute, OpenDocument represents the cells on either side of the large cell as *sub-tables*. Figure 4.6, “Cells Spanning Rows” shows a table with a cell that spans two rows. As far as OpenDocument is concerned, the table has only two rows. The second row consists of:

- A cell that contains a two-by-one subtable
- An ordinary cell (labelled `main 2,2`)
- A cell that spans two columns and contains a two-by-two subtable.

Figure 4.6. Cells Spanning Rows



main 1,1	main 1,2	main 1,3	main 1,4
left 1,1	main 2,2	right 1,1	right 1,2
left 2, 1		right 2,1	right 2,2

Example 4.7, “XML for Cells Spanning Rows” shows the relevant XML for the second row of this table named `vertSpan`. Most of the `text:style-name` and all of the `office:value-type` attributes removed for ease of reading. We’ve also added comments within the listing.

Example 4.7. XML for Cells Spanning Rows

```
<table:table-row>
    <table:table-cell>
        <table:table table:is-sub-table="true">
            <table:table-column table:style-name="vertSpan.A"/>

            <table:table-row>
                <!-- notice how the style name is constructed
                    for a sub-table cell -->
                <table:table-cell table:style-name="vertSpan.A2.1.1"
                  office:value-type="string">
                    <text:p>left 1,1</text:p>
                </table:table-cell>
            </table:table-row>

            <table:table-row>
                <table:table-cell>
                    <text:p>left 2, 1</text:p>
                </table:table-cell>
            </table:table-row>
        </table:table>
    </table:table-cell>
    <table:table-cell table:style-name="vertSpan.B2">
        <text:p>main 2,2</text:p>
    </table:table-cell>

    <table:table-cell table:number-columns-spanned="2">
        <table:table table:is-sub-table="true">
            <table:table-column table:number-columns-repeated="2"/>

            <table:table-row>
                <table:table-cell table:style-name="vertSpan.C2.1.1">
                    <text:p>right 1,1</text:p>
                </table:table-cell>
                <table:table-cell table:style-name="vertSpan.C2.2.1">
                    <text:p>right 1,2</text:p>
                </table:table-cell>
            </table:table-row>

            <table:table-row>
                <table:table-cell>
                    <text:p>right 2,1</text:p>
                </table:table-cell>
                <table:table-cell>
                    <text:p>right 2,2</text:p>
                </table:table-cell>
            </table:table-row>
        </table:table> <!-- ends subtable -->

    </table:table-cell>

    <!-- NOTE the covered-table-cell is required because
    the right sub-table has covered the rightmost column -->
    <table:covered-table-cell/>
</table:table-row>
```

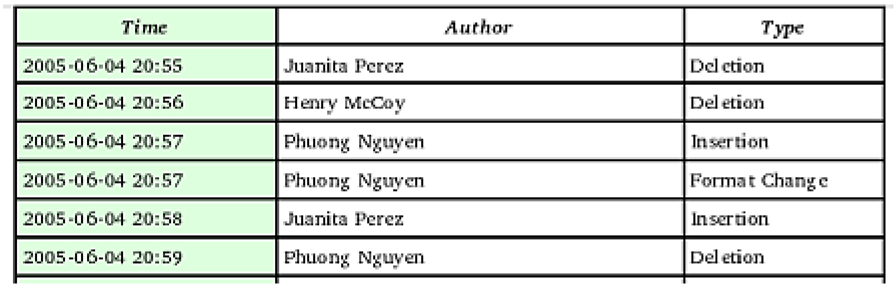
Case Study: Creating a Table of Changes

Let’s put this information to use by creating a document that contains a table that summarizes the changes made in another OpenDocument file. We will use XSLT to do this transformation.

Figure 4.7, “Change Summary, Sorted by Time” shows a portion of some sample output, reduced and cropped to save space. It was produced by running the XSL transformation, `changetable.xsl`, on file `changetest.odt`, which you can find in directory `ch04` in the downloadable example files.

The table will contain three columns: the time, author, and type of change. It can be sorted by any of the three columns, and the column that is used for the sort is highlighted in light green. The transformation accepts a parameter named `sort` with the value of `time`, `author`, or `type` to specify the sorting criterion.

Figure 4.7. Change Summary, Sorted by Time



Time	Author	Type
2005-06-04 20:55	Juanita Perez	Deletion
2005-06-04 20:56	Henry McCoy	Deletion
2005-06-04 20:57	Phuong Nguyen	Insertion
2005-06-04 20:57	Phuong Nguyen	Format Change
2005-06-04 20:58	Juanita Perez	Insertion
2005-06-04 20:59	Phuong Nguyen	Deletion

The stylesheet begins with an `<xsl:stylesheet>` that provides all the relevant namespaces and an `<xsl:output>` element that sets the output method to XML and turns on indenting. These can be copied straight from Example C.6, "XSLT Framework for Transforming OpenDocument" and are not shown here.

Here's the XSLT to set up the "outer structure" of the output document.

```
<xsl:template match="/">
<office:document-content
xmlns:office="http://openoffice.org/2000/office"
    xmlns:style="http://openoffice.org/2000/style"
    xmlns:text="http://openoffice.org/2000/text"
    xmlns:fo="http://www.w3.org/1999/XSL/Format"
    xmlns:table="http://openoffice.org/2000/table"
    office:class="text">
    <office:scripts/>

    <office:font-face-decls>
        <style:font-face style:name="Bitstream Charter"
            svg:font-family="&apos;Bitstream Charter&apos;"
            style:font-pitch="variable"/>
    </office:font-face-decls>

    <office:automatic-styles> ❶
        <style:style style:name="P1" style:family="paragraph">
            <style:text-properties style:font-name="Bitstream Charter"
            fo:font-size="10pt" style:font-size-asian="10pt"
            style:font-size-complex="10pt"/>
        </style:style>

        <style:style style:name="P2" style:family="paragraph">
            <style:paragraph-properties fo:text-align="center"/>
            <style:text-properties style:font-name="Bitstream Charter"
            fo:font-size="10pt" style:font-size-asian="10pt"
            style:font-size-complex="10pt"
            fo:text-align="center"
            fo:font-style="italic"
            fo:font-weight="bold"/>
        </style:style>

        <style:style style:name="ctable" style:family="table"> ❷
            <style:table-properties
        style:width="15cm" table:align="center" />
</style:style>

<style:style style:name="ctable.A"
   style:family="table-column">
    <style:table-column-properties
      style:column-width="4.5cm" />
</style:style>

<style:style style:name="ctable.B"
  style:family="table-column">
    <style:table-column-properties style:column-width="7cm"/>
</style:style>

<style:style style:name="ctable.C"
  style:family="table-column">
    <style:table-column-properties
      style:column-width="3.5cm"/>
</style:style>

<style:style style:name="ctable.A1"
  style:family="table-cell"> ❸
    <style:table-cell-properties
        fo:border-top="0.035cm solid #000000"
        fo:border-right="none"
        fo:border-bottom="0.035cm solid #000000"
        fo:border-left="0.035cm solid #000000"
        fo:padding="0.10cm">
        <xsl:call-template name="set-bg-color"> ❹
            <xsl:with-param
                name="col-type">time</xsl:with-param>
        </xsl:call-template>
    </style:table-cell-properties>
</style:style>

<style:style style:name="ctable.B1" style:family="table-cell">
    <style:table-cell-properties
        fo:border-top="0.035cm solid #000000"
        fo:border-right="none"
        fo:border-bottom="0.035cm solid #000000"
        fo:border-left="0.035cm solid #000000"
        fo:padding="0.10cm">
        <xsl:call-template name="set-bg-color">
            <xsl:with-param
                name="col-type">author</xsl:with-param>
        </xsl:call-template>
    </style:table-cell-properties>
</style:style>

<style:style style:name="ctable.C1" style:family="table-cell">
    <style:table-cell-properties
        fo:border="0.035cm solid #000000"
        fo:padding="0.10cm">
        <xsl:call-template name="set-bg-color">
            <xsl:with-param
                name="col-type">type</xsl:with-param>
        </xsl:call-template>
    </style:table-cell-properties>
</style:style>
```

```
        <style:style style:name="ctable.A2"
          style:family="table-cell"> ❺
            <style:table-cell-properties
                fo:border-top="none"
                fo:border-right="none"
                fo:border-bottom="0.035cm solid #000000"
                fo:border-left="0.035cm solid #000000"
                fo:padding="0.10cm">
                <xsl:call-template name="set-bg-color">
                    <xsl:with-param name="col-type">time</xsl:►
with-param>
                </xsl:call-template>
            </style:table-cell-properties>
        </style:style>

        <style:style style:name="ctable.B2" style:family="table-cell">
            <style:table-cell-properties
                fo:border-top="none"
                fo:border-right="none"
                fo:border-bottom="0.035cm solid #000000"
                fo:border-left="0.035cm solid #000000"
                fo:padding="0.10cm">
                <xsl:call-template name="set-bg-color">
                    <xsl:with-param name="col-type">author</xsl:►
with-param>
                </xsl:call-template>
            </style:table-cell-properties>
        </style:style>

        <style:style style:name="ctable.C2" style:family="table-cell">
            <style:table-cell-properties
                fo:border-top="none"
                fo:border-right="0.035cm solid #000000"
                fo:border-bottom="0.035cm solid #000000"
                fo:border-left="0.035cm solid #000000"
                fo:padding="0.10cm">
                <xsl:call-template name="set-bg-color">
                    <xsl:with-param name="col-type">type</xsl:►
with-param>
                </xsl:call-template>
            </style:table-cell-properties>
        </style:style>

    </office:automatic-styles>

    <office:body>
        <office:text>
            <table:table table:name="ctable"
              table:style-name="ctable"> ❻
                <table:table-column table:style-name="ctable.A" />
                <table:table-column table:style-name="ctable.B" />
                <table:table-column table:style-name="ctable.C" />
                <table:table-header-rows>
                    <table:table-row>
                        <table:table-cell table:style-name="ctable.A1"
                            office:value-type="string">
                            <text:h text:style-name="P2">Time</text:h>
                        </table:table-cell>
                        <table:table-cell table:style-name="ctable.B1"
                            office:value-type="string">
                            <text:h
                                text:style-name="P2">Author</text:h>
                        </table:table-cell>
                        <table:table-cell table:style-name="ctable.C1"
                            office:value-type="string">
                            <text:h text:style-name="P2">Type</text:h>
                        </table:table-cell>
                    </table:table-row>
                </table:table-header-rows>

                <xsl:choose> ❼
                    <xsl:when test="$sort = 'time' or
                        $sort = 'author'">
                        <xsl:apply-templates
                            select="office:►
document-content/office:body/
                            office:text/text:tracked-changes"/>
                    </xsl:when>
                    <xsl:otherwise>
                        <xsl:apply-templates
                            select="office:►
document-content/office:body/
                            office:text/text:tracked-changes/
                            text:changed-region[text:insertion]"/>
                        <xsl:apply-templates
                            select="office:►
document-content/office:body/
                            office:text/text:tracked-changes/
                            text:changed-region[text:deletion]"/>
                        <xsl:apply-templates
                            select="office:►
document-content/office:body/
                            office:text/text:tracked-changes/
                            text:changed-region[text:format-change]"/>
                    </xsl:otherwise>
                </xsl:choose>
            </table:table>
        </office:text>
    </office:body>
</office:document-content>
</xsl:template>
```

❶ Style `P1` will be used for all text except the table headings, which use style `P2`, which makes headings bold, italic, and centered.

❷ The table is 15 centimeters wide, with columns of 4.5, 7, and 3.5 centimeters.

❸ Because we will be adding a background color to only one cell in each row, we have to create separate styles for each cell in a row. Styles `ctable.A1`,`ctable.B1`, and `ctable.C1` are for the cells in the first row. Note that only `ctable.C1` has a right border.

❹ This template will add the background color to the style if the global `sort` parameter (specified by the user) matches the `col-type`, which is the "type of data this column contains."

❺ We have to create a similar set of styles for the second and subsequent rows; none of these has a top border (since the bottom margin of the row above fills in that line) and, again, only `ctable.C2` has a right border.

❻ The table begins with the three `<table:table-column>` elements, followed by the first row of the table. The first row is enclosed in a `<table:table-header-rows>` element, so it will be repeated in case the table extends across a page boundary.

❼ If the user decides to sort by time or author, all we have to do is hand all the `<text:tracked-changes>` elements off to a template and let it do the sorting by the values of the `<dc:date>` or `<dc:creator>`. If the user decides to sort by type, then we process all the insertions first, then the deletions, then the format changes.

Here is the template that sets the background color attribute if the column name given to it as a parameter is the column we are sorting on:

```
<xsl:template name="set-bg-color">
    <xsl:param name="col-type"/>
    <xsl:if test="$sort = $col-type">
        <xsl:attribute
            name="fo:background-color">#ddffdd</xsl:attribute>
    </xsl:if>
</xsl:template>
```

The following template handles the sorting of time or date; notice that we must search the `descendant::` axis, since the `<office:change-info>` element is a grandchild of the `<text:changed-region>`.

```
<xsl:template match="text:tracked-changes">
    <xsl:choose>
    <xsl:when test="$sort = 'time'">
        <xsl:apply-templates select="text:changed-region">
            <xsl:sort
                select="descendant::office:change-info/dc:date"/>
        </xsl:apply-templates>
    </xsl:when>
    <xsl:when test="$sort = 'author'">
        <xsl:apply-templates select="text:changed-region">
            <xsl:sort
                select="descendant::office:change-info/dc:creator"/>
        </xsl:apply-templates>
    </xsl:when>
    </xsl:choose>
</xsl:template>
```

Each `<text:changed-region>` creates a new table row, with the appropriate data in each cell. We call a template named `format-time` to change the ISO8601 format to something slightly less unpleasant.

```
<xsl:template match="text:changed-region">
    <table:table-row>
        <table:table-cell table:style-name="ctable.A2">
            <text:p text:style-name="P1">
                <xsl:call-template name="format-time">
                    <xsl:with-param name="time"
                        select="descendant::office:►
change-info/dc:date"/>
                </xsl:call-template>
            </text:p>
        </table:table-cell>

        <table:table-cell table:style-name="ctable.B2">
            <text:p text:style-name="P1">
                <xsl:value-of
                    select="descendant::office:►
change-info/dc:creator"/>
            </text:p>
        </table:table-cell>

        <table:table-cell table:style-name="ctable.C2">
            <text:p text:style-name="P1">
                <xsl:choose>
                    <xsl:when test="text:insertion">
                        <xsl:text>Insertion</xsl:text>
                    </xsl:when>
                    <xsl:when test="text:deletion">
                        <xsl:text>Deletion</xsl:text>
                    </xsl:when>
                    <xsl:when test="text:format-change">
                        <xsl:text>Format Change</xsl:text>
                    </xsl:when>
                </xsl:choose>
            </text:p>
        </table:table-cell>
    </table:table-row>
</xsl:template>
```

Here is the time formatter; it simply removes the `T` from the time, and drops the seconds from the time of day.

```
<xsl:template name="format-time">
    <xsl:param name="time"/>
    <xsl:value-of select="substring-before($time, 'T')"/>
    <xsl:text> </xsl:text>
    <xsl:value-of select="substring(substring-after($time,
       'T'),1,5)"/>
</xsl:template>
```

The stylesheet ends with a template that will eliminate any stray text nodes from the output:

```
<xsl:template match="text()"/>
```

Chapter 5. Spreadsheets

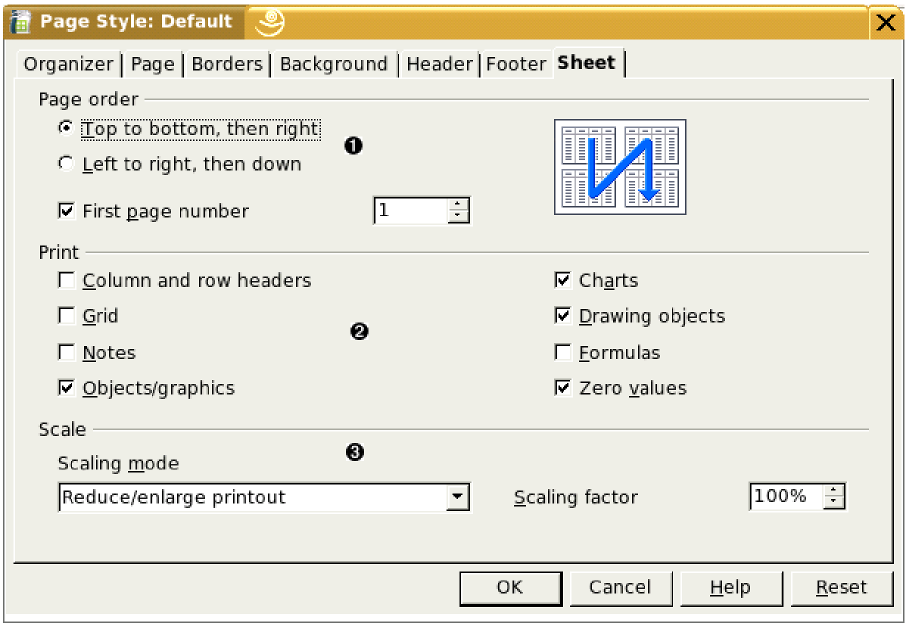
Surprisingly, we have already covered a great deal of the information about spreadsheets. Spreadsheets share a great deal of their markup with tables that you find in text documents. This shouldn't come as a surprise—a spreadsheet is just a two-dimensional table. It can have many blank rows and columns and can do calculations on the cell entries, but a spreadsheet is still just a table at heart.

However, there are things that make a spreadsheet, well, spreadsheetish. Most important, the `<office:body>` has an `<office:spreadsheet>` element as its child (rather than `<office:text>` for a word processing document). Other elements and attributes specific to spreadsheets are in the `styles.xml` file, but most are in `content.xml`.

Spreadsheet Information in styles.xml

The `styles.xml` file stores information that OpenOffice.org sets from the sheet tab of the Format Page dialog, shown in Figure 5.1, "Spreadsheet Page Options". Specifically, this information is in the `<style:page-layout-properties>` element that is inside the first `<style:page-layout>` element within the `<office:automatic-styles>`.

Figure 5.1. Spreadsheet Page Options



Page order
[✓ Top to bottom, then right]
Left to right, then down
❶
[✓] First page number: 1

Print
[ ] Column and row headers
[ ] Grid
[ ] Notes
[✓] Objects/graphics
❷
[✓] Charts
[✓] Drawing objects
[ ] Formulas
[✓] Zero values

Scale
❸
Scaling mode: Reduce/enlarge printout
Scaling factor: 100%

1. The `style:print-page-order` attribute has a value of `ttb` for top to bottom, and `ltr` for left to right. If the first page number is not one (the default), then the `style:first-page-number` attribute will give the number that you specify.
2. The value of the `style:print` attribute summarizes all the marked checkboxes as a whitespace-separated list. If you turn on all the checkboxes, the value will be these words (separated by whitespace): `annotations`, `charts`, `drawings`, `formulas`, `grid`, `headers`, `objects`, and `zero-values`.
3. If you are scaling to a percentage, then the `style:scale-to` attribute will have the scaling percentage (with a percent sign) as its value. If you are fitting to a number of pages, then the `style:scale-to-pages` attribute will provide that value. If you are scaling to width and height, then the `style:scale-to-X` and `style:scale-to-Y` attributes will give the number of pages in each direction.

Example 5.1, "Page Options" shows this markup.

Example 5.1. Page Options

```
<office:automatic-styles>
    <style:page-layout style:name="pm1">
        <style:page-layout-properties
            style:print-page-order="ttb" style:first-page-number="2"
            style:scale-to-pages="1"
            style:writing-mode="lr-tb"
            style:print="formulas grid headers zero-values"/>
        <!-- etc. -->
    </style:page-master>
    <!--  more styles -->
</office:automatic-styles>
```

Spreadsheet Information in content.xml

The `<office:automatic-styles>` element contains

- Column styles
- Row styles
- Sheet styles
- Number styles
- Cell Styles

Column and Row Styles

Each differently styled column in the spreadsheet gets a `<style:style>` whose `style:family` is `table-column`. Its child `<style:table-column-properties>` element specifies the width of the column (`style:column-width`) in the form of a *length*, such as `1.1in`.

The column styles are followed by `<style:style>` elements whose `style:family` is `table-row`. Their child `<style:table-row-properties>` element specifies the `style:row-height`. If you have chosen “optimal height” then this element will also set `style:use-optimal-row-height` to `true`.

Styles for the Sheet as a Whole

A `<style:style>` element with a `style:family="table"` primarily serves to connect a table with a master page and to determine whether the sheet is hidden or not. Example 5.2, “Style Information for a Sheet” shows just such an element.

Example 5.2. Style Information for a Sheet

```
<style:style style:name="ta1" style:family="table"
    style:master-page-name="Default">
    <style:table-properties table:display="true"/>
</style:style>
```

Number Styles

The other major style information in a spreadsheet deals with formatting numbers so that they show up as percentages, scientific notation, etc. A number style must convey two pieces of information: what kind of a number it is (number, currency, percent, date, etc.) and how the number is to be displayed. This is mirrored in the XML as a `<number:`*`entity`*`-style>` element, where *entity* can be `number`, `currency`, `percent`, `date`, etc.

This element has a required `style:name` attribute that gives the style a unique identifier, and a `style:family` attribute with a value of `data-style`. The contents of this element will tell how to display the number, percent, currency, date, etc.

Number, Percent, Scientific, and Fraction Styles

Let’s start with the “pure numeric” styles: numbers, percents, scientific notation, and fractions.

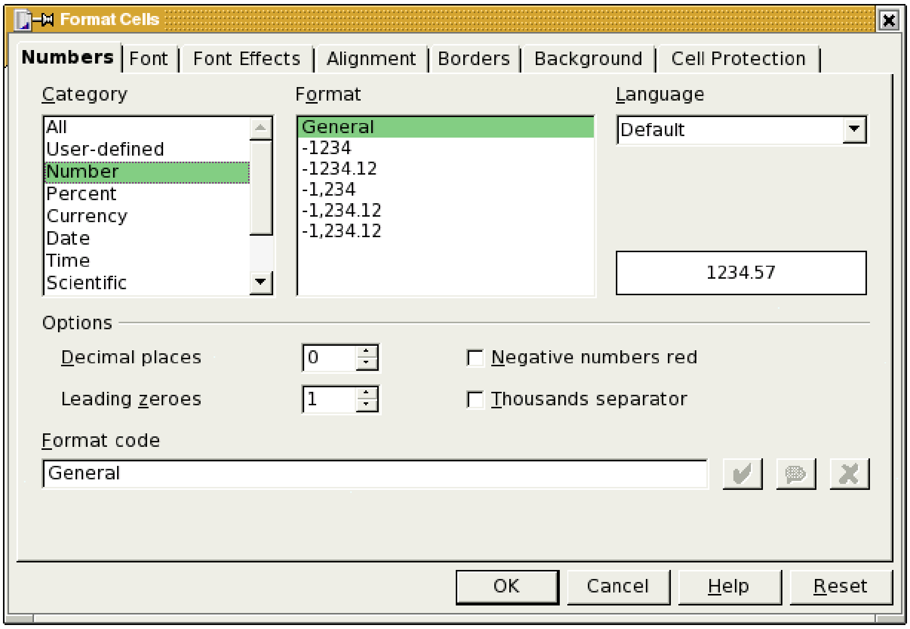
Plain Numbers

A plain number is contained in a `<number:number-style>` element with a `style:name` attribute. Contained within this element is the description of how to display the number. In this case, we need only a simple `<number:number>` element that has these attributes

- `number:decimal-places` tells how many digits are to the right of the decimal symbol
- `number:min-integer-digits` tells how many leading zeros are present.

- `number:grouping`. If you have checked the“thousands separator” dialog item, then this attribute will be present and will have a value of `true`.

Figure 5.2. Number Styles Dialog



Category: All, User-defined, Number, Percent, Currency, Date, Time, Scientific (Number selected)

Format: General, -1234, -1234.12, -1,234, -1,234.12, -1,234.12 (General selected)

Language: Default

1234.57

Options

Decimal places: 0

Leading zeroes: 1

☐ Negative numbers red

☐ Thousands separator

Format code: General

Example 5.3, “Number Style for format #,##0.00” shows a number style for displaying two places to the right of the decimal, one leading zero, and a grouping separator.

Example 5.3. Number Style for format #,##0.00

```
<number:number-style style:name="N2"">
    <number:number number:decimal-places="2"
            number:min-integer-digits="1"
        number:grouping="true"/>
</number:number-style>
```

Note

The decimal symbol and grouping symbol are not specified in the style; they are set in the application.

If you want negative numbers to be red, then things become radically different. Rather than having one style, OpenDocument requires two styles, with the negative being the default and a “conditional style” for positive values. Here is the XML for a number with two digits to the right of the decimal, one leading zero, a thousands separator, and negative numbers in red:

Example 5.4. Number Style for format -#,##0.00 with Negative Values in Red

```
<number:number-style style:name="N112P0"
    style:volatile="true"> ❶
    <number:number number:decimal-places="2"
        number:min-integer-digits="1" number:grouping="true"/>
</number:number-style>

<number:number-style style:name="N112">
    <style:text-properties fo:color="#ff0000"/>  ❷
    <number:text>-</number:text>  ❸
    <number:number number:decimal-places="2"  ❹
        number:min-integer-digits="1" number:grouping="true"/>
    <style:map style:condition="value()&gt;=0"  ❺
        style:apply-style-name="N112P0"/>
</number:number-style>
```

❶ This is the format to be used for positive numbers. The `style:volatile="true"` tells the application to retain this style, even if it is never used.

❷ This is the main style for negative numbers. They should be displayed in red …

❸ … starting with a minus sign …

❹ … followed by the number with two decimal places, at least one leading zero, and a thousands separator.

❺ However, in the event that the value of the cell is greater than or equal to (`&gt;=`) zero, use the positive number style (`N112P0`).

Scientific Notation

Scientific notation is a variant on plain numbers; the outer `<number:number-style>` contains a `<number:scientific-number>` element with these attributes: `number:decimal-places` and `number:min-integer-digits` for the mantissa, and `number:min-exponent-digits` for the exponent part. You don't need to put the `E` in the specification. Example 5.5, "Scientific Notation for Format 0.00E+00" shows the style for scientific notation with two digits to the right of the decimal point, at least one to the left, and at least two digits in the exponent.

Example 5.5. Scientific Notation for Format 0.00E+00

```
<number:number-style style:name="N3">
    <number:scientific-number
        number:decimal-places="2"
        number:min-integer-digits="1"
        number:min-exponent-digits="2"/>
</number:number-style>
```

Fractions

Fractions are also variants of plain numbers. Their `<number:number-style>` element contains a `<number:fraction>` element that has these attributes: `number:min-integer-digits` (number of digits in the whole number part), `number:min-numerator-digits`, and `number:min-denominator-digits`. Example 5.6, "Fraction Style for Format # ??/??" shows a style for a fraction with an optional whole number part and at least two digits in the numerator and denominator.

Example 5.6. Fraction Style for Format # ??/??

```
<number:number-style style:name="N4">
    <number:fraction
        number:min-integer-digits="0"
        number:min-numerator-digits="2"
        number:min-denominator-digits="2"/>
</number:number-style>
```

Percentages

A percentage is represented in much the same way as a plain number; the only differences are that

- The enclosing element is `<number:percentage-style>` instead of `<number:number-style>`.
- The enclosed `<number:number>` style is followed by a `<number:text>` element with a percent sign as its content.

Example 5.7, "Percent Style for Format #,##0.00%" shows a percentage with two digits to the right of the decimal, at least one to the left, and a grouping symbol.

Example 5.7. Percent Style for Format #,##0.00%

```
<number:percentage-style style:name="N5">
    <number:number number:decimal-places="2"
        number:min-integer-digits="1" number:grouping="true"/>
    <number:text>%</number:text>
</number:percentage-style>
```

Currency Styles

Currency styles are similar to number styles. Specifying a currency always creates two styles: one for negative values, and one for positive values. Example 5.8, "Currency in Format -$#,##0.00" shows the XML for a currency format of US Dollars with two digits after the decimal point, a minimum of one digit before the decimal, and a thousands separator.

Example 5.8. Currency in Format -$#,##0.00

```
<number:currency-style style:name="N102P0" ❶
    style:volatile="true">
    <number:currency-symbol  ❷
        number:language="en"
        number:country="US">$</number:currency-symbol>
    <number:number number:decimal-places="2"  ❸
        number:min-integer-digits="1" number:grouping="true"/>
</number:currency-style>

<number:currency-style style:name="N102">
    <number:text>-</number:text>  ❹
    <number:currency-symbol number:language="en"
        number:country="US">$</number:currency-symbol>
    <number:number number:decimal-places="2"
        number:min-integer-digits="1" number:grouping="true"/>
    <style:map style:condition="value()&gt;=0"
        style:apply-style-name="N102P0"/>  ❺
</number:currency-style>
```

❶ The formatting for positive values appears first, contained in a `<number:currency-style>` element.

❷ The `<number:currency-symbol>` 's content is the dollar sign. The `number:language` and `number:country` allow a program to distinguish the US dollar from the New Zealand dollar or Mexican new peso symbol, which look the same but have different countries and/or languages.

❸ The number portion of the display uses the same `<number:number>` element that we have already described[6].

❹ For negative values, the minus sign precedes the currency symbol.

❺ As in Example 5.4, “Number Style for format -#,##0.00 with Negative Values in Red”, a `<style:map>` is used to choose whether to use the negative number format or the positive number format.

The appearance of `<number:text>` elements mirrors the order in which the text appears. Example 5.9, “Currency Format for Greek Drachma” shows the negative number portion of the XML for the Greek drachma. In this format, the value is shown in red, the minus sign appears first, then the number, then a blank and the letters “Δρχ.” (We are showing only the negative number specification.)

[6] If you want to have a replacement for the decimal part of the number (as in `$15.--`), you add `number:decimal-replacement="--"` to the `<number:number>` element.

Example 5.9. Currency Format for Greek Drachma

```
<number:currency-style style:name="N111">
    <style:properties fo:color="#ff0000"/>
    <number:text>-</number:text>
    <number:number number:decimal-places="2"
        number:min-integer-digits="1" number:grouping="true"/>
    <number:text> </number:text>
    <number:currency-symbol number:language="el"
        number:country="GR">Δρχ</number:currency-symbol>
    <style:map style:condition="value()&gt;=0"
        style:apply-style-name="N111P0"/>
</number:currency-style>
```

Date and Time Styles

OpenDocument applications support a large number of different formats for dates and times. Rather than explain each one in detail, it's easier to simply compose the style you want out of parts.

For dates, the enclosing element is a `<number:date-style>` element, with the usual `style:name` attribute. The `number:automatic-order` attribute is used to automatically order data to match the default order for the language and country of the data. You may also set the `number:format-source` to `fixed`, to let the application determine the value of "short" and "long" representations of months, days, etc. If the value is `language`, then those values are taken from the language and country set in the style.

Within the `<number:date-style>` element are the following elements, with their significant attributes:

`<number:year>`
: Gives the year in two-digit form; the year 2003 appears as 03. If `number:style="long"` then the year appears as four digits.

`<number:month>`
: If `number:textual="true"` then the month appears as an abbreviated name; otherwise a number without a leading zero. To get the full name of the month or the month number with a leading zero, set `number:style="long"`.

`<number:day-of-week>`
: The day of the week as an abbreviated name; to see the full name, use `number:style="long"`.

`<number:day>`
: The day of the month as a number without a leading zero; to see leading zeros, use `number:style="long"`.

`<number:quarter>`
: Which quarter of the year; in U.S. English, a date in October appears as `Q4`. If `number:style="long"`, then it appears as `4th quarter`.

`<number:week-of-year>`
: Displays which week of the year this date occurs in; thus January 1st displays as `1` and December 31st displays as `52` (or, in OpenOffice.org's case, as `1` if there are 53 weeks in the year, as there are in 2003!)

Example 5.10, "Date Styles" shows three date styles. The first will display the fourth day of the seventh month of 2005 as `Monday, July 4, 2005`; the second will display it as `07/04/05`, and the third as `3rd Quarter 05`.

Example 5.10. Date Styles

```
<number:date-style style:name="N79" number:automatic-order="true">
    <number:day-of-week number:style="long"/>
    <number:text>, </number:text>
    <number:month number:style="long"
        number:textual="true"/>
    <number:text> </number:text>
    <number:day/>
    <number:text>, </number:text>
    <number:year number:style="long"/>
</number:date-style>

<number:date-style style:name="N37" number:automatic-order="true">
    <number:month number:style="long"/>
    <number:text>/</number:text>
    <number:day number:style="long"/>
    <number:text>/</number:text>
    <number:year/>
</number:date-style>

<number:date-style style:name="N20106">
    <number:quarter number:style="long"/>
    <number:text> </number:text>
    <number:year/>
</number:date-style>
```

Time values are represented by the `<number:time-style>` element. Its sub-elements are:

`<number:hours>`
: Shows the number of hours; if you want leading zeros on hours less than ten, set `number:style="long"`. If a duration is more than 24 hours, it will be displayed mod 24. If you do not want this to happen, then set `number:truncate-on-overflow="false"` *on the* `<number:time-style>` *element.*

`<number:minutes>`
: Displays the number of minutes without a leading zero; if you want two digits, set `number:style="long"`.

`<number:seconds>`
: Displays the number of seconds without a leading zero; if you want two digits, set `number:style="long"`. If you wish to see decimal fractions of a second, then add a `number:decimal-places` attribute whose value is the number of decimal places you want.

`<number:am-pm>`
: This empty element inserts the appropriate `am` or `pm` (in the selected locale).

Example 5.11, “Time Style” shows the style required to display a time in the format `09:02:34 AM`

Example 5.11. Time Style

```
<number:time-style style:name="N43">
    <number:hours number:style="long"/>
    <number:text>:</number:text>
    <number:minutes number:style="long"/>
    <number:text>:</number:text>
    <number:seconds number:style="long"/>
    <number:text> </number:text>
    <number:am-pm/>
</number:time-style>
```

Note

A `<number:date-style>` element may also specify hours, minutes, and seconds.

Internationalizing Number Styles

An OpenDocument-compatible application gets its cues for displaying numbers from the current language setting. You may set the display of a number to a specific language and country by adding the `number:language` and `number:country` attributes to a `<number:`*`entity`*`-style>` element. Thus, to make a date display in Korean format, you would start the specification as follows:

```
<number:date-style style:name="N5076"
  number:language="ko" number:country="KR">
    <number:year number:style="long"/>
    <number:text>년 </number:text>
    <number:month/>
    <number:text>월 </number:text>
    <number:day/>
    <number:text>일</number:text>
</number:date-style>
```

Cell Styles

Finally, each different style of cell has its own `<style:style>` element. If the cell contains text, then it will contain a `<style:text-properties>` element that describes its border, background color, font, alignment, etc. If it contains a number, then the style contains a reference to one of the previously established number styles. Example 5.12, "Style for a Numeric Cell" shows the XML for the cell containing the time style shown in Example 5.11, "Time Style".

Example 5.12. Style for a Numeric Cell

```
<style:style style:name="ce8" style:family="table-cell"
    style:parent-style-name="Default"
    style:data-style-name="N43"/>
```

Table Content

Let us now turn our attention to the table content, which is contained in `content.xml`, inside the `<office:body>` element. Each sheet is stored as a separate `<table:table>`. Its `table:name` attribute is the name that will appear on the spreadsheet tab, and the `table:style-name` attribute refers to a table style as described in the section called "Styles for the Sheet as a Whole".

Columns and Rows

The `<table:table>` element contains a series of `<table:table-column>` elements to describe each of the columns in the table. These each have a `table:style-name` attribute whose value refers to a `<style:style>` with that name. If several consecutive columns all have the same style, then a `table:number-columns-repeated` attribute tells how many times it is repeated. A hidden column will have its `table:visibility` attribute set to `collapse`.

Example 5.13, "Table Columns in a Spreadsheet" shows the XML for the columns of a table with eight columns. The second and last columns have the same style, and there are three identical columns before the last one.

Example 5.13. Table Columns in a Spreadsheet

```
<table:table-column table:style-name="co1"
    table:default-cell-style-name="ce1" />
<table:table-column table:style-name="co2"
    table:default-cell-style-name="Default" />
<table:table-column table:style-name="co3"
    table:default-cell-style-name="ce2" />
<table:table-column table:style-name="co4"
    table:number-columns-repeated="3"
    table:default-cell-style-name="Default" />
<table:table-column table:style-name="co2"
    table:default-cell-style-name="Default" />
```

The column specifications are followed by the `<table:table-row>` elements. These also have a `table:style-name` attribute referring to a `<style:style>` with a `style:family="table-row"`. If the row is duplicated, then the `table:number-rows-repeated` gives the repetition count. A hidden row has `table:visibility` set to `collapse`.

String Content Table Cells

Within the table row are the `<table:table-cell>` entries. If the cell contains a string, then the cell will contain a child `<text:p>` element that contains the text, as in the following example:

```
<table:table-cell>
    <text:p>Federico Gutierrez</text:p>
</table:table-cell>
```

Numeric Content in Table Cells

Cells that contain numbers also contain a `<text:p>` that shows the *display form* of the value. The actual value is stored in the `<table:table-cell>` element with two attributes: `office:value-type` and `office:value`. These are related as described in Table 5.1, "office:value-type and office:value".

Table 5.1. office:value-type and office:value

office:value-type	**office:value**
`float`	Used for pure numbers, fractions, and scientific notation. The value is stored without a decimal point if the value is an integer; otherwise `.` is used as the decimal point.
`percentage`	A display value of `45.6%` is stored as `0.456`.
`currency`	The value is stored using `.` as a decimal point, with no currency symbol. There is an additional `table:currency` attribute that contains an abbreviation such as `USD` or `GRD`.
`date`	The value is stored in a `office:date-value` attribute rather than a `office:value`. If it contains a simple date, it is stored in the form `yyyy-mm-dd`; if there is both a day and a time, it is stored in the form `yyyy-mm-ddThh:mm:ss`.
`time`	The value is stored in a `office:time-value` attribute rather than a `office:value`. The value is stored in the form `PThhHmmMss,ffffS` (where `ffff` is the fractional part of a second).

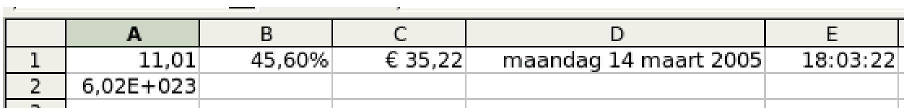
Note

The content of the `<text:p>` element is provided as a convenience for programs that wish to harvest the displayed values. OpenOffice.org will display cell contents based upon the `office:value` and `office:value-type` *only*, ignoring the content of the cell's `<text:p>`.

Putting it all Together

Figure 5.3, “Spreadsheet Showing Various Data Types” shows a simple spreadsheet with the default language set to Dutch (Netherlands).

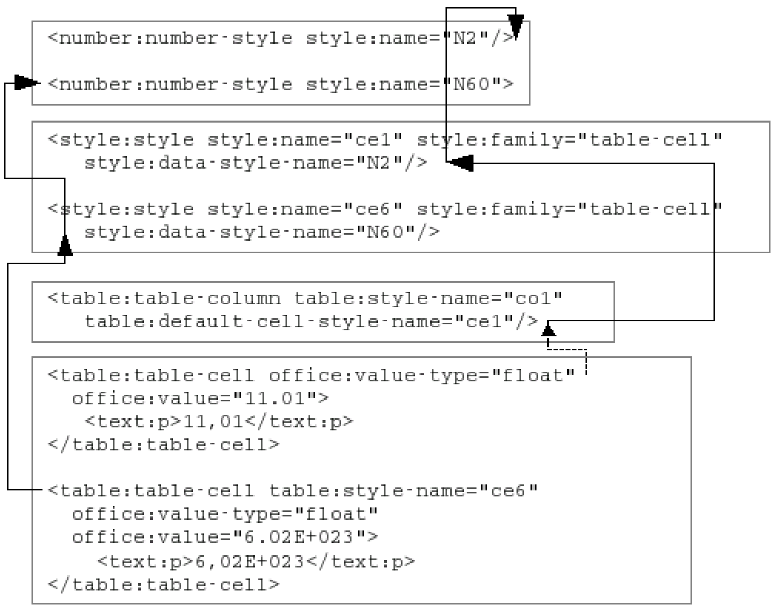
Figure 5.3. Spreadsheet Showing Various Data Types



	A	B	C	D	E
1	11,01	45,60%	€ 35,22	maandag 14 maart 2005	18:03:22
2	6,02E+023				

Showing you the actual XML would be more confusing than illuminating. Instead, we’ve boiled down the linkage to Figure 5.4, “Spreadsheet Showing Number Style Linkages”, starting at a table cell.

Figure 5.4. Spreadsheet Showing Number Style Linkages



```
<number:number-style style:name="N2"/>

<number:number-style style:name="N60">
```

```
<style:style style:name="ce1" style:family="table-cell"
   style:data-style-name="N2"/>

<style:style style:name="ce6" style:family="table-cell"
   style:data-style-name="N60"/>
```

```
<table:table-column table:style-name="co1"
   table:default-cell-style-name="ce1"/>
```

```
<table:table-cell office:value-type="float"
  office:value="11.01">
    <text:p>11,01</text:p>
</table:table-cell>

<table:table-cell table:style-name="ce6"
  office:value-type="float"
  office:value="6.02E+023">
    <text:p>6,02E+023</text:p>
</table:table-cell>
```

- If you have a `table:style-name`, then that’s the style for that cell.
- If you don’t have a `table:style-name`, then the column this cell is in leads you indirectly to the style via its corresponding `<table:table-column>`.
- In either case, you end up at a `<style:style>` element whose `style:data-style-name` attribute leads you to …
- A `<number:number-style>` that tells you how the cell should be formatted.

Formula Content in Table Cells

Formula cells contain a `table:formula` attribute. Within the `table:formula` attribute, references to individual cells or cell ranges are enclosed in square brackets. Relative cell names are expressed in the form *sheetname*`.`*`cellname`*. Thus, a reference to cell A3 in the current spreadsheet will appear as `[.A3]`, and a reference to cell G17 in a spreadsheet named `Sheet2` will appear as `[Sheet2.G17]`. The range of cells from G3 to K7 in the current spreadsheet appear as `[.G3:.K7]`.

Absolute cell names simply have the preceding `$` on them, much as you would enter them in OpenOffice.org. Thus, an absolute reference to cell C4 in the current spreadsheet would be written as `[.$C$4]`.

Depending upon the return type of the formula, the table cell will contain appropriate `office:value` and `office:value-type` attributes. Example 5.14, "Return Types from Formulas" shows the result of three formulas; the first returns a simple number, the second returns a string showing roman numerals, and the third produces a time value from the contents of three cells.

Example 5.14. Return Types from Formulas

```
<table:table-cell table:formula="oooc:=SUM([.A1:.C1])"
  office:value-type="float" office:value="137">
    <text:p>137</text:p>
</table:table-cell>

<table:table-cell table:formula="oooc:=ROMAN([.B4])"
  office:value-type="string" office:string-value="CVII">
    <text:p>CVII</text:p>
</table:table-cell>

<table:table-cell table:formula="oooc:=TIME([.E1];[.E2];[.E3])"
  office:value-type="time" office:time-value="PT10H05M48S">
    <text:p>10:05:48 AM</text:p>
</table:table-cell>
```

According to the specification, an OpenDocument-compatible application should depend *only* upon the formula to generate its display. A program could generate a spreadsheet that would display identically to the preceding example when opened in OpenOffice.org, using only the information shown in Example 5.15, "Minimal Formulas".

Example 5.15. Minimal Formulas

```
<table:table-cell table:formula="oooc:=SUM([.A1:.C1])"/>

<table:table-cell table:formula="oooc:=ROMAN([.B4])"/>

<table:table-cell table:formula="oooc:=TIME([.E1];[.E2];[.E3])"/>
```

If you are using an array formula which would be represented in OpenOffice.org within curly braces, such as {=B6:C6*B7:C7}, you must specify the number of rows and columns that the result will occupy. The preceding formula is marked up as follows:

```
<table:table-cell
    table:number-matrix-columns-spanned="2"
    table:number-matrix-rows-spanned="1"
    table:formula="oooc:=[.B6:.C6]*[.B7:.C7]"
    office:value-type="float" office:value="27">
    <text:p>27</text:p>
</table:table-cell>
```

Merged Cells in Spreadsheets

Merging cells in spreadsheets is far easier than merging them in text tables. The first cell in the merged area will have `table:number-rows-spanned` and `table:number-columns-spanned` attributes. Their values give the number of rows and columns that have been merged. Any of the cells which have been covered by the merged cell will no longer be ordinary `<table:cell>` elements; they will become `<table:covered-table-cell>` elements, but the rest of their attributes and contents will remain unchanged.

Case Study: Modifying a Spreadsheet

We will use this information about spreadsheets to write a Python program that does currency conversion. All cells that are stored in one currency (such as U.S. dollars) will be converted to the equivalent values in a different currency (such as Korean Won) and saved to a new spreadsheet.

To find and change the appropriate `<number:currency-style>` elements, the program must know the values of `number:country` and `number:language` for the source and destination currencies. To find and change the appropriate `<table:table-cell>` elements, the program must know the three-letter abbreviation found in `table:currency` for the source and destination currencies.

Finally, we will need to provide format strings for positive and negative values in the destination currency, the currency symbol for the destination currency, and a conversion factor for multiplying the value of the numbers in the spreadsheet. We will store all this information in an ad-hoc XML file of the form shown in Example 5.16, "Money Conversion Parameters", which converts U.S. dollars to Korean Won. [This is file `currencyparam.xml` in directory `ch05` in the downloadable example files.]

Example 5.16. Money Conversion Parameters

```
<convert>
    <from language="en" country="US" abbrev="USD" />
    <to language="ko" country="KR" abbrev="KRW"
        symbol="₩"
        positiveFormat="$#,##0.00"
        negativeFormat="-$#,##0.00"
        factor="1175.19"/>
</convert>
```

The symbols in the format string have the following meanings:

- `$` represents the currency symbol (as described by the `symbol` attribute).
- `#` represents a digit other than a leading or trailing zero.
- `,` means that this number has a thousands separator.
- `0` represents a digit including leading and trailing zeros.
- `.` represents the decimal point (which will be displayed in the appropriate locale within the application).

All the other characters in the format string are taken as text. This allows you to place blanks and other characters in a format.

Main Program

Although Python requires functions to be defined before they are used, we are doing a top-down explanation of this program, so we will present functions in conceptual order rather than file order. Here's the main program, which looks for three arguments on the command line: the filename of the OpenDocument file, the filename for the resulting document, and the filename of the parameter XML file. [The main program is file `currency_conversion.py` in directory `ch05` in the downloadable example files.]

```
import xml.dom
import xml.dom.ext
import xml.dom.minidom
import xml.parsers.expat
import sys
import od_number
from zipfile import *
from StringIO import *

if (len(sys.argv) == 4):

    #   Open an existing OpenDocument file
    #
    inFile = ZipFile( sys.argv[1] )

    #   ...and a brand new output file
    #
    outFile = ZipFile( sys.argv[2], "w", ZIP_DEFLATED );

    getParameters( sys.argv[3] )

    #
    #   modify all appropriate currency styles
    #
    fixCurrency( "styles.xml" )
    fixCurrency( "content.xml" )

    #
    #   copy the manifest
    #
    copyManifest( )

    inFile.close
    outFile.close
else:
    print "Usage: " + sys.argv[0] + " inputfile outputfile
parameterfile"
```

The import statements give us access to various Python libraries. `od_number` is a module that we have written to convert format strings to XML document fragments.

Getting Parameters

Here is the code that will open the parameter file and read the data into global variables.

```
def getParameters( filename ):
    global oldLanguage, oldCountry, oldAbbrev
    global language, country, abbreviation, currencySymbol
    global positiveFormatString, negativeFormatString, factor

    paramFile = open( filename, "r" ) ❶
    document = xml.dom.minidom.parse( paramFile )

    node = document.getElementsByTagName( "from" )[0] ❷
    oldLanguage = node.getAttribute( "language" )
    oldCountry = node.getAttribute( "country" )
    oldAbbrev = node.getAttribute( "abbrev" )

    node = document.getElementsByTagName( "to" )[0]
    language = node.getAttribute( "language" )
    country = node.getAttribute( "country" )
    abbreviation = node.getAttribute( "abbrev" )
    currencySymbol = node.getAttribute( "symbol" )
    positiveFormatString = node.getAttribute( "positiveFormat" )
    negativeFormatString = node.getAttribute( "negativeFormat" )
    factor = float( node.getAttribute("factor") ) ❸

    paramFile.close()
```

❶ Creating a DOM tree is easy with Python; just open a file and feed it to the `parse` function.

❷ Even though there is only one `<from>` element and `<to>` element in the parameter file, `getElementsByTagName` always returns a node list. No problem—we'll just grab item `[0]` from that list.

❸ All the other parameters are string values, but the multiplication factor is a number, so we use `float` to convert from string to numeric.

Converting the XML

Take a deep breath and hold on tight; this is the largest function in the program.

```
def fixCurrency( filename ):

    #
    #   Read the styles.xml file as a string file
    #   and create a disk file for output
    #
    dataSource = StringIO (inFile.read( filename ))  ❶
    tempFileName = "/tmp/workfile"
    dataSink = open(tempFileName, "w")

    #
    #   Parse the document
    #
    document = xml.dom.minidom.parse( dataSource )

    #
    #   Create document fragments from the format strings  ❷
    #
    posXML = od_number.ODNumber( document, positiveFormatString,
        language, country, currencySymbol )
    posXML.createCurrencyStyle( )

    negXML = od_number.ODNumber( document, negativeFormatString,
        language, country, currencySymbol )
    negXML.createCurrencyStyle( )

    #
    #   Fix number style elements
    #
    currencyElements = document.getElementsByTagName►
("number:currency-symbol")
    for element in currencyElements:
        if (element.getAttribute( "number:language" ) == oldLanguage►
and  ❸
            element.getAttribute( "number:country" ) == oldCountry):

            element.setAttribute( "number:language", language )
            element.setAttribute( "number:country", country )

            parent = element.parentNode  ❹
            children = parent.childNodes
            i = len(children)-1
            while (i >= 0):
                if (children[i].nodeName == "number:number" or
                children[i].nodeName == "number:text" or
                children[i].nodeName == "number:currency-symbol" or
                children[i].nodeType == xml.dom.Node.TEXT_NODE):
                    parent.removeChild( children[i] )
                i = i - 1

            #   select the appropriate number format markup
            if ((parent.getAttribute("style:name"))[-2:] == "P0"):  ❺
                fragment = posXML.getFragment()
            else:
                fragment = negXML.getFragment()
            #
            #   and insert it into the <number:currency-style> element
            for child in fragment.childNodes: ❻
                parent.appendChild( child.cloneNode(True) )


    #
    #   Fix table cells (which only exist in content.xml) ❼
    #
    rowElements = document.getElementsByTagName("table:table-row")
    for row in rowElements:
        cell = getChildElement( row, "table:table-cell" ) ❽
        while (cell != None):
            if (cell.getAttribute("table:currency") ==  oldAbbrev ):

                # change the currency abbreviation
                cell.setAttribute("table:currency", abbreviation )

                # and the number in the cell, if there is one
                valueStr = cell.getAttribute("office:value")
                if (valueStr != ""):
                    result = float( valueStr ) * factor
                    cell.setAttribute("office:value",  '%f' % result)

                #   remove any children of this cell
                children = cell.childNodes ❾
                i = len(children)-1
                while (i >= 0):
                    cell.removeChild( children[i] )
                    i = i - 1

            # move to the next cell in the row
            cell = getSiblingElement( cell, "table:table-cell" )

    #
    #   Serialize the document tree to the output file
    xml.dom.ext.Print( document, dataSink )
    dataSink.close();

    #
    #   Add the temporary file to the new .zip file, giving it
    #   the same name as the input file.
    #
    outFile.write( tempFileName, filename )
```

❶ The input file is a member of a .zip file; we can't pass the .zip file itself on to the parser. Nor can we open a file descriptor for a member of the .zip archive, so we are forced to read in the input file into a string, and use the `StringIO` constructor to make it look like a file.

On the other hand, we can't easily write a string to a member of the output file, so we create a temporary file on disk. (The filename is a Unix filename; change it as appropriate for your system.)

❷ We will convert the format strings to document fragments so that we can just copy the XML from the fragments into the DOM tree that we are modifying. This is nontrivial code, so it's separated out into another module altogether.

❸ We don't want to indiscriminately modify all the `<number:currency-symbol>` elements; you may have multiple currencies in your document, and you want to change only the ones specified in your parameters.

❹ Before we put in the new format markup, we have to get rid of the old markup. We don't eliminate all the old stuff; we want to keep any `<style:properties>` (for red text) and `<style:map>` elements (which select positive or negative formats).

When removing the children, we have to go in reverse order; if we had started by removing child number zero, then child number one would move into its place and we would miss it on the next loop iteration.

❺ This code presumes that you are using a file that has been created with OpenOffice.org; currency formats for positive values always end with the characters `P0`.

❻ This code uses the `cloneNode()` function to make sure that all of the fragment nodes' descendants get copied into the document being modified.

❼ Rather than retrieve all the `<table:table-cell>` elements at once, which could strain memory with a large document, we get cells one row at a time.

❽ We can't just go to the first child of the table row; there may be intervening whitespace text nodes. Thus, we have our own `getChildElement()` function to find the node we really want. A similar `getSiblingElement()` function finds the next sibling while avoiding those pesky whitespace nodes.

❾ Rather than try to update the value of the `<text:p>` inside the cell (which would force us to do all the calculation and formatting that OpenOffice.org does), we just eliminate it and let OpenOffice.org re-create it after a load and save.

Copying the manifest also creates a temporary file:

```
def copyManifest():
    #
    #   read the manifest.xml file as a string
    #   and create a disk file for transfer to the .zip output
    dataSource = inFile.read( "META-INF/manifest.xml" )
    tempFileName = "/tmp/workfile"
    dataSink = open(tempFileName, "w")
    dataSink.write( dataSource );
    dataSink.close();
    outFile.write( tempFileName, "META-INF/manifest.xml" )
```

DOM Utilities

These are the utility functions that we mentioned in the preceding section. They search for the first child or next sibling of a node that has the desired element name, while avoiding any extraneous text nodes.

```
def getChildElement( node, name ):
    node = node.firstChild;
    while (node != None):
        if (node.nodeType != xml.dom.Node.ELEMENT_NODE or
          node.nodeName != name):
            node = node.nextSibling
        else:
            break
    return node

def getSiblingElement( node, name ):
    node = node.nextSibling;
    while (node != None):
        if (node.nodeType != xml.dom.Node.ELEMENT_NODE or
          node.nodeName != name):
            node = node.nextSibling
        else:
            break
    return node
```

Parsing the Format Strings

Finally, the code for parsing the format string to produce an XML document fragment.

I put this into a module, even though it's not something that would be useful for any other program. I did it this way because I didn't know of any other way to do an "include." Hey, this is my first Python program of any size greater than "Hello, World!" [This module is file `od_number.py` in directory `ch05` in the downloadable example files.]

```
import xml.dom
import xml.dom.ext
import xml.dom.minidom
import re

class ODNumber:
    def __init__(self, document, formatString, language, country,
    currencySymbol ):
        self.thousands = False;      # thousands separator?
        self.nDecimals = 0;          # number of decimal places
        self.minIntegerDigits = 0;   # min. integer digits
        self.textStr = ""            # text string being built
        self.fragment = None         # fragment being built

        self.document = document     # copy parameters to class ►
attributes
        self.language = language
        self.country = country
        self.currencySymbol = currencySymbol
        self.formatString = formatString
    def endStr( self ):  ❶
        if (self.textStr != ""):
            textElement = self.document.createElement( "number:text" )
            textNode = self.document.createTextNode( self.textStr )
            textElement.appendChild( textNode )
            self.fragment.appendChild( textElement )
            self.textStr = ""

    def addCurrency( self ):
        self.endStr() ❷
        node = self.fragment.appendChild(
            self.document.createElement( "number:currency-symbol" ) )
        node.setAttribute( "number:language", self.language )
        node.setAttribute( "number:country", self.country )
        node.appendChild( self.document.createTextNode( ►
self.currencySymbol ) )

    def addNumber( self ): ❸
        node = self.fragment.appendChild(
            self.document.createElement("number:number") )
        node.setAttribute( "number:min-integer-digits",
            "%d" % self.minIntegerDigits )
        if (self.nDecimals > 0):
            node.setAttribute( "number:decimal-places",
                "%d" % self.nDecimals )
        if (self.thousands):
            node.setAttribute( "number:grouping", "true" )

    def createCurrencyStyle ( self ):
        """Scan a format string, where:

        $ indicates the currency symbol
        # indicates an optional digit
        0 indicates a required digit
        , indicates the thousands separator (no matter your locale)
        . indicates the decimal point (no matter your locale)

        Creates a document fragment with appropriate OpenOffice.org
        markup.
        """

        self.fragment = self.document.createElement("number:fragment")❹

        hasDecimal = False

        numchars = re.compile( "[#,0.]" )   ❺

        i = 0
        while (i < len(self.formatString)):
            char = self.formatString[i]

            if (char == "$"):
                self.addCurrency( )
            elif (re.search( numchars, char )):
                self.endStr( ) ❻
                while (re.search( numchars, char )):

                    if (char == ","):
                        self.thousands = True
                    elif (char == "0"):
                        if (hasDecimal):
                            self.nDecimals = self.nDecimals + 1
                        else:
                            self.minIntegerDigits = \
                                self.minIntegerDigits + 1
                    elif (char == "."):
                        hasDecimal = True;

                    if (i == len(self.formatString) - 1):
                        break;
                    i = i + 1
                    char = self.formatString[i]
                self.addNumber( )
            else:
                self.textStr = self.textStr + char ❼
            i = i + 1
        self.endStr( ) ❽

    def getFragment( self ): ❾
        return self.fragment
```

❶ As we parse a string, anything that isn't part of the format string or currency symbol gets accumulated into the `textStr` variable. This function dumps it out to an `<number:text>` element.

❷ If we have accumulated any text prior to the currency symbol (it could be a minus sign), output it.

❸ This function is called when we reach the end of the number part of the format string.

❹ We create a phony element to be the container for all the other elements that we are going to create. Why should we go to the trouble of creating our own tree structure to hold independently-created nodes when that's part of the DOM's job?

❺ This compiles a regular expression; matching against it is easier than doing a large conditional expression to see if we have a character that's part of the number format.

❻ As soon as we find one of those crucial characters, we output any pending text, then gather the format information in the `while` loop.

❼ If the character isn't a currency symbol or part of a number format, it's just generic text to be accumulated.

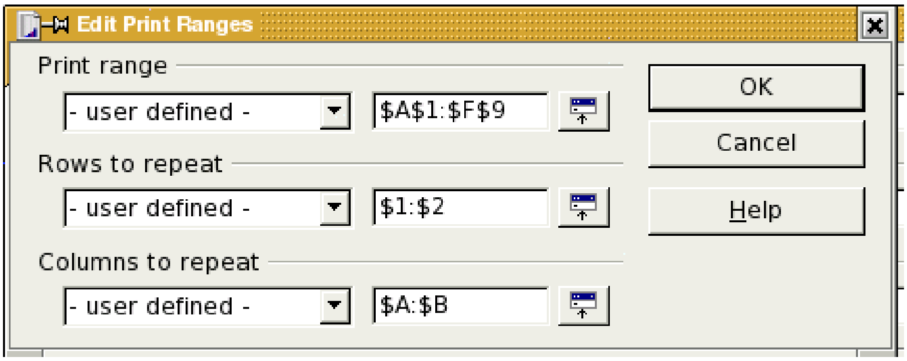
❽ If there's any pending text when we hit the end of the string, we need to put it into the output.

❾ A utility function to return the fragment that's been built. Truth in advertising: Since this module wasn't designed for object-oriented purity, it is here more for appearance's sake than anything else.

Print Ranges

If you wish to specify a print range for the sheet (corresponding to the dialog box shown in Figure 5.5, "Spreadsheet Print Ranges" add a `table:print-ranges` attribute to the `<table:table>` element. Its value will be in a form like `Sheet1.A1:Sheet1:F9`.

Figure 5.5. Spreadsheet Print Ranges


Edit Print Ranges

Print range: - user defined - $A$1:$F$9

Rows to repeat: - user defined - $1:$2

Columns to repeat: - user defined - $A:$B

The `<table:table-column>` elements that are to be repeated will be enclosed in a `<table:table-header-columns>` element; the `<table:table-row>` elements to be repeated will be enclosed in a `<table:table-header-rows>` element.

Example 5.17, "Structure of Print Ranges" shows the skeleton of the XML markup for the print ranges chosen in Figure 5.1, "Spreadsheet Page Options".

Example 5.17. Structure of Print Ranges

```
<table:table table:name="Sheet1" table:style-name="ta1"
    table:print-ranges="Sheet1.A1:Sheet1.F9">

   <table:table-header-columns>
       <table:table-column table:style-name="co1"/>
       <table:table-column table:style-name="co2"/>
    </table:table-header-columns>

    <table:table-column table:style-name="co3"/>
    <table:table-column table:style-name="co4"/>
    <!-- remaining non-header columns -->

    <table:table-header-rows>
        <table:table-row table:style-name="ro1">
            <table:table-cell>
                <text:p>Cell A1</text:p>
            </table:table-cell>
            <table:table-cell>
                <!-- rest of row 1 -->
            </table:table-cell>
        </table:table-row>
        <table:table-row table:style-name="ro2">
            <table:table-cell>
                <text:p>Cell A2</text:p>
            </table:table-cell>
            <table:table-cell>
                <!-- rest of row 2 -->
            </table:table-cell>
        </table:table-row>
    </table:table-header-rows>

    <table:table-row table:style-name="ro3">
        <table:table-cell><text:p>Cell A3</text:p></table:table-cell>
        <table:table-cell> <!-- rest of row 1 --> </table:table-cell>
    </table:table-row>

    <!-- remaining non-header rows -->

</table>
```

Case Study: Creating a Spreadsheet

Our task in this case study is to use XSLT to transform data from an XML-based student gradebook and convert it to an OpenDocument spreadsheet. This is the actual markup that I use for the classes that I teach, and it is the actual transformation that I use.

The source XML document's root element is a `<gradebook>` element. It contains a `<task-list>`, which gives information about each `<task>` that has been assigned to a student. Each task has an `id` attribute, `date` the assignment was due, a `max` (maximum) possible score, a `type` (lab, quiz, midterm, etc.), a `weight` telling what percentage of the final score this task is worth, and a `recorded` attribute that tells whether the scores for this task have been recorded or not. For example, I always have a midterm exam which I enter in the task list at the beginning of the semester; I just don't set its `recorded` flag until the midterm has been given.

Following the task list is a series of `<student>` elements, each of which has an `id` attribute (the social security number preceded by a letter `S`. The `<student>` element contains the student's last and first names, email address, extra info, and a series of `<result>` elements.

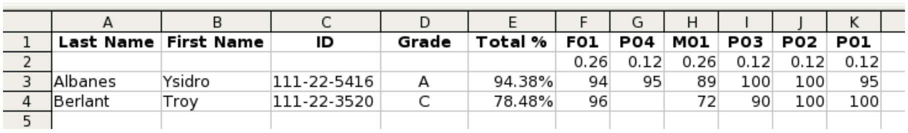
Each `<result>` element has a `score` attribute and a `ref` attribute. This last attribute is a reference to a task id from the task list. If I have some comments about the student's work, that becomes the text content of the `<result>` element. Example 5.18, "Sample Gradebook Data" shows part of a gradebook. No real students or social security numbers were harmed in creating this data. [This is file `minigrades.xml` in directory `ch05` in the downloadable example files.]

Example 5.18. Sample Gradebook Data

```
<gradebook>
    <task-list>
        <task date="2003-02-07" id="P01" max="100" recorded="yes"
            type="lab" weight="12">Program 1 (Functions)</task>
        <task date="2003-02-21" id="P02" max="100" recorded="yes"
            type="lab" weight="12">Program 2 (Window and►
 Document)</task>
        <task date="2003-03-07" id="P03" max="100" recorded="yes"
            type="lab" weight="12">Program 3 (Forms)</task>
         <task date="2003-04-04" id="M01" max="100" recorded="yes"
            type="midterm" weight="26">Midterm</task>
        <task date="2003-05-09" id="P04" max="100" recorded="yes"
            type="lab" weight="12">Program 4 (DHTML)</task>
        <task date="2003-05-24" id="F01" max="100" recorded="yes"
            type="final" weight="26">Final</task>
    </task-list>
    <student id="S111-22-5416">
        <last>Albanes</last>
        <first>Ysidro</first>
        <email>albanes.ysidro@example.com</email>
        <info/>
        <result ref="P01" score="95"/>
        <result ref="P02" score="100">Good work!</result>
        <result ref="P03" score="100"/>
        <result ref="M01" score="89"/>
        <result ref="P04" score="95"/>
        <result ref="F01" score="94"/>
    </student>
    <student id="S111-22-3520">
        <last>Berlant</last>
        <first>Troy</first>
        <email>berlant.troy@example.com</email>
        <info/>
        <result ref="P01" score="100"/>
        <result ref="P02" score="100"/>
        <result ref="P03" score="90"/>
        <result ref="M01" score="72"/>
        <result ref="P04" score=""/>
        <result ref="F01" score="96"/>
    </student>
    <!-- etc. -->
</gradebook>
```

The resulting spreadsheet should look like Figure 5.6, “Result of Gradebook Transformation”. The first row has the headings, with tasks labelled by their task ID. The second row shows the weights for each task as decimals (a weight of 12 becomes 0.12), and the remaining rows show the results for each student.

Figure 5.6. Result of Gradebook Transformation



	A	B	C	D	E	F	G	H	I	J	K
1	**Last Name**	**First Name**	**ID**	**Grade**	**Total %**	**F01**	**P04**	**M01**	**P03**	**P02**	**P01**
2						0.26	0.12	0.26	0.12	0.12	0.12
3	Albanes	Ysidro	111-22-5416	A	94.38%	94	95	89	100	100	95
4	Berlant	Troy	111-22-3520	C	78.48%	96		72	90	100	100
5											

The XSLT file is fairly long, so we will look at it in parts. The first part establishes namespaces, includes a file with some standard font declarations, as described in the section called “Font Declarations”, and sets the course name in a parameter. The parameter allows users to enter the course name from the command line. [The entire XSLT file is file `gradebook_to_ods.xsl` in directory `ch05` in the downloadable example files.]

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns:office="urn:oasis:names:tc:opendocument:xmlns:office:1.0"
    xmlns:meta="urn:oasis:names:tc:opendocument:xmlns:meta:1.0"
    xmlns:config="urn:oasis:names:tc:opendocument:xmlns:config:1.0"
    xmlns:text="urn:oasis:names:tc:opendocument:xmlns:text:1.0"
    xmlns:table="urn:oasis:names:tc:opendocument:xmlns:table:1.0"
    xmlns:draw="urn:oasis:names:tc:opendocument:xmlns:drawing:1.0"
    xmlns:presentation="urn:oasis:names:tc:opendocument:xmlns:►
presentation:1.0"
    xmlns:dr3d="urn:oasis:names:tc:opendocument:xmlns:dr3d:1.0"
    xmlns:chart="urn:oasis:names:tc:opendocument:xmlns:chart:1.0"
    xmlns:form="urn:oasis:names:tc:opendocument:xmlns:form:1.0"
    xmlns:script="urn:oasis:names:tc:opendocument:xmlns:script:1.0"
    xmlns:style="urn:oasis:names:tc:opendocument:xmlns:style:1.0"
    xmlns:number="urn:oasis:names:tc:opendocument:xmlns:datastyle:1.0"
    xmlns:anim="urn:oasis:names:tc:opendocument:xmlns:animation:1.0"

    xmlns:dc="http://purl.org/dc/elements/1.1/"
    xmlns:xlink="http://www.w3.org/1999/xlink"
    xmlns:math="http://www.w3.org/1998/Math/MathML"
    xmlns:xforms="http://www.w3.org/2002/xforms"

    xmlns:fo="urn:oasis:names:tc:opendocument:xmlns:►
xsl-fo-compatible:1.0"
    xmlns:svg="urn:oasis:names:tc:opendocument:xmlns:►
svg-compatible:1.0"
    xmlns:smil="urn:oasis:names:tc:opendocument:xmlns:►
smil-compatible:1.0"

    xmlns:ooo="http://openoffice.org/2004/office"
    xmlns:ooow="http://openoffice.org/2004/writer"
    xmlns:oooc="http://openoffice.org/2004/calc"
>

<xsl:output method="xml" indent="yes"/>

<xsl:include href="fontdecls.xsl"/>

<xsl:param name="courseName">Course</xsl:param>
```

The next line builds a “lookup table” of all the `<student>` nodes in the document, indexed by the value of their `id` attribute. We will use this index, which has the name `student-index`, to access a student’s information from another node’s context.

```
<xsl:key name="student-index" match="student" use="@id"/>
```

Here's the first template: what to output when we encounter the root node of the document. It will create all the styles and begin the body of the document. We gave the cell styles meaningful names rather than using the OpenOffice.org naming convention (`ce1`, `ce12`, etc), because it helped us keep track of which cells had which style. OpenOffice.org doesn't care what you name your styles, as long as they are all referenced correctly; this should be the case with all OpenDocument compatible applications.

Note the order in which the styles occur: column styles, row styles, number styles, and then cell styles.

```
<office:document-content office:version="1.0">

<office:scripts/>

<xsl:call-template name="insert-font-decls"/>

<office:automatic-styles>

    <!-- Column for last and first name -->
    <style:style style:name="co1" style:family="table-column">
        <style:table-column-properties fo:break-before="auto"
            style:column-width="2.5cm"/>
    </style:style>

    <!-- column for final grade and percentage -->
    <style:style style:name="co2" style:family="table-column">
        <style:table-column-properties fo:break-before="auto"
            style:column-width="2cm"/>
    </style:style>

    <!-- All other columns -->
    <style:style style:name="co3" style:family="table-column">
        <style:table-column-properties fo:break-before="auto"
            style:column-width="1.25cm"/>
    </style:style>

    <!-- Let all the rows have optimal height -->
    <style:style style:name="ro1" style:family="table-row">
        <style:table-row-properties fo:break-before="auto"
            style:use-optimal-row-height="true"/>
    </style:style>

    <!-- The table references a master-page which doesn't exist,
        but that doesn't bother OpenOffice.org -->
    <style:style style:name="ta1" style:family="table"
        style:master-page-name="TAB_Sheet1">
        <style:table-properties table:display="true"/>
    </style:style>

    <!-- Number style for a percentage -->
    <number:percentage-style style:name="N01">
        <number:number number:decimal-places="2"
            number:min-integer-digits="1"/>
        <number:text>%</number:text>
    </number:percentage-style>

    <!-- individual cell styles -->
    <!-- style for final grade letter -->
    <style:style style:name="centered" style:family="table-cell"
        style:parent-style-name="Default">
        <style:paragraph-properties fo:text-align="center"/>
    </style:style>

    <!-- style for the total grade percent -->
    <style:style style:name="percent" style:family="table-cell"
        style:data-style-name="N01"/>

    <!-- style for heading cells -->
    <style:style style:name="heading" style:family="table-cell"
        style:parent-style-name="Default">
        <style:paragraph-properties fo:text-align="center"/>
        <style:text-properties fo:font-weight="bold"/>
    </style:style>

    <!-- style for raw data cells (just use OpenOffice.org defaults)
-->
    <style:style style:name="normal" style:family="table-cell"
        style:parent-style-name="Default"/>
</office:automatic-styles>

<office:body>
    <office:spreadsheet>
```

We now proceed to the spreadsheet content, starting with the column specifications.

```
<!-- calculate number of raw data columns -->
<xsl:variable name="numTasks" ❶
    select="count(gradebook/task-list/task[@recorded='yes'])"/>

<!-- start the spreadsheet -->
<table:table table:name="{$courseName} Final Grades"
  table:style-name="ta1">  ❷

    <!-- last name, first name, and ID are repeated when printing -->
    <table:table-header-columns>  ❸
        <table:table-column table:style-name="co1"
            table:default-cell-style-name="normal"/>
        <table:table-column table:style-name="co1"
            table:default-cell-style-name="normal"/>
        <table:table-column table:style-name="co1"
            table:default-cell-style-name="normal"/>
    </table:table-header-columns>

    <!-- final grade -->
    <table:table-column table:style-name="co2"
        table:default-cell-style-name="centered" />
    <!-- percentage -->
    <table:table-column table:style-name="co2"
        table:default-cell-style-name="percent" />

    <!-- everyone else -->
    <table:table-column table:style-name="co3"
        table:default-cell-style-name="normal"
        table:number-columns-repeated="{$numTasks}"> ❹
    </table:table-column>
```

❶ Calculate the number of columns in the spreadsheet by counting the number of tasks that have been recorded.

❷ We want the value of the `courseName` parameter to become the value of the `table:table-name` attribute. The `$` gets the value of the variable; the braces tell XSLT to evaluate the contents as an XPath expression rather than the literal string `$courseName`.

❸ The first three columns should appear on every page if the printout is more than one page long, so their specifications are enclosed in a `<table:table-header-columns>` element.

❹ All the raw data columns have the same format, so we use the `table:number-columns-repeated` attribute, setting its value to the value of (that's what `$` stands for) the `numTasks` variable.

Having specified the columns, we continue to the rows, which contain the actual table data. The first row contains headings, all of which have the `heading` style (bold and centered).

```
<table:table-header-rows> ❶
    <table:table-row table:style-name="ro1">
        <table:table-cell table:style-name="heading">
            <text:p>Last Name</text:p>
        </table:table-cell>

        <table:table-cell table:style-name="heading">
            <text:p>First Name</text:p>
        </table:table-cell>

        <table:table-cell table:style-name="heading">
            <text:p>ID</text:p>
        </table:table-cell>

        <table:table-cell table:style-name="heading">
            <text:p>Grade</text:p>
        </table:table-cell>

        <table:table-cell table:style-name="heading">
            <text:p>Total %</text:p>
        </table:table-cell>

        <!-- emit heading cell for each recorded task -->
        <xsl:for-each ❷
            select="gradebook/task-list/task[@recorded='yes']">
            <xsl:sort select="@date" order="descending"/> ❸
                <table:table-cell table:style-name="heading">
                    <text:p><xsl:value-of select="@id"/></text:p>
                </table:table-cell>
        </xsl:for-each>
    </table:table-row>
</table:table-header-rows>
```

- ❶ The first row should appear on every page if the printout is more than one page long, so its specification is enclosed in a `<table:table-header-rows>` element.
- ❷ `<xsl:for-each>` is XSLT's only iterative structure; it selects all the specified nodes and then applies all the processing contained within it.
- ❸ The `<xsl:sort>` modifies the `<xsl:for-each>` by specifying the order in which the selected nodes are to be processed. Rather than processing tasks in document order, we process them in descending order by their `date` attribute.

The next row contains the weights for each task. The first five cells will be unoccupied, and we use another `<xsl:for-each>` to place the weights into the remaining cells. Note that the calculation of the `factor` variable uses `div` for division. That is because the forward slash symbol is already used to separate steps in an XPath expressions, and using it for division as well would complicate life for everyone.

```
<table:table-row table:style-name="ro1">
    <!-- five empty cells -->
    <table:table-cell table:number-columns-repeated="5"/>
    <xsl:for-each select="gradebook/task-list/task[@recorded='yes']">
        <xsl:sort select="@date" order="descending"/>

        <xsl:variable name="factor" select="@weight div @max"/>

        <table:table-cell office:value-type="float"
            office:value="{$factor}">
            <text:p><xsl:value-of select="$factor"/></text:p>
        </table:table-cell>
    </xsl:for-each>
</table:table-row>
```

The only remaining job is to create the rows for each student; we will handle this by applying a template to all the `<student>` elements.

```
            <!-- Create one row for each student -->
            <xsl:apply-templates select="gradebook/student">
                <xsl:sort select="last"/> ❶
                <xsl:sort select="first"/>
                <xsl:with-param name="numCols" select="$numTasks+5"/> ❷
            </xsl:apply-templates>
        </table:table>
    </office:spreadsheet>
</office:body>
</office:document-content>
</xsl:template>
```

- ❶ In this case, we are using `<xsl:sort>` to modify the order in which the templates are applied to the selected nodes. The first `<xsl:sort>` specifies the primary sort key; the second `<xsl:sort>` specifies the secondary key. You may have as many keys as you need.

❷ When we create the formula for calculating the student's grade, we have to know the ending column number for the row. This is the number of tasks plus five, so we send that to the template as a parameter named `numCols`.

Now we provide the template that handles the processing of the `<student>` element selected in the preceding code.

```
<xsl:template match="student">
    <xsl:param name="numCols"/> ❶

    <!-- generate the column letter for the last column -->
    <xsl:variable name="lastCol">
        <xsl:call-template name="generate-col">  ❷
            <xsl:with-param name="n" select="$numCols"/>
        </xsl:call-template>
    </xsl:variable>
```

❶ Any template which accepts parameters via `<xsl:with-param>` must declare the parameter name.

❷ The code for generating the letter corresponding to the last column number is non-trivial; rather than placing it here inside the template, we have put it in a template named `generate-col` (which does not correspond to any element name in our document). You can think of these named templates as subroutines rather than as templates that match document elements. That is why we call it via `<xsl:call-template>` instead of `<xsl:apply-templates>`. We are going to call the template with a parameter `n` (`<xsl:with-param>`). The parameter will be the number of columns in this template; its value will be the same as `numCols`. In the case of six tasks, the `lastCol` variable will end up with the value `K`.

The code for the first name, last name, and student ID cells is quite straightforward; it uses the `substring()` function to eliminate the letter `S` from the student ID.

Warning

In XSLT, the first character in a string is character number one, not number zero, as you would find in most programming languages.

```
<!-- cells for last name, first name, and student ID -->
<table:table-cell>
    <text:p><xsl:value-of select="last"/></text:p>
</table:table-cell>
<table:table-cell>
    <text:p><xsl:value-of select="first"/></text:p>
</table:table-cell>
<table:table-cell>
    <text:p><xsl:value-of select="substring(@id,2)"/></text:p>
</table:table-cell>
```

The next columns are the final letter grade and the total percentage. Extra whitespace has been inserted into the following listing to make things easier to read.

```
<!-- formula for final grade letter -->
<table:table-cell
    table:formula=
    "oooc:=MID(&quot;FFFFFFDCBAA&quot;; INT([.E{position()+2}]*10)►
+1; 1)"/>

<!-- formula for final grade as a percentage -->
<table:table-cell
    table:number-matrix-columns-spanned="1"
    table:number-matrix-rows-spanned="1"
    table:formula=
        "oooc:=SUM([.F{position()+2}:.{$lastCol}{position()+2}] *
        [.$F$2:.${$lastCol}$2])/100"
    office:value-type="float"/>
```

The letter grade formula works by looking at the column to its right (the percentage), multiplying it by 10 to give a number between 0 and 10, then grabbing the corresponding letter from the string `FFFFFFDCBAA`. It uses the `position()` function to figure out which row number is being output. Here's how that function works: `<xsl:apply-templates>` has selected a set of student nodes and `<xsl:sort>` has sorted them. Each one in turn is being processed by the current `<xsl:template>`. The first one to be processed has a `position()` equal to 1, the second has a position of 2, etc. Since there are two rows of headings in the spreadsheet, the row number being output is the `position()` plus two. Thus, for the first student, the formula works out to `oooc:=MID("FFFFFFDCBAA";INT([.E3]*10)+1;1)` (the `oooc:` does not show up in the application).

The next cell also uses the `position()` function and the `lastCol` variable to create the correct formula. Because the cell contains an array formula, it needs the `table:number-matrix-columns-spanned` and `table:number-matrix-rows-spanned` attributes to specify the dimensions of the resulting array. If the template is processing the first student and there are six tasks, then the XSLT creates the formula that OpenOffice.org displays as `{=SUM([.F3:.K3]*[.$F$2:.$K$2])/100}`.

The following is the remainder of the template for handling a student. The lines have been numbered for reference.

```
 1  <!-- save the student's id -->
 2  <xsl:variable name="id" select="@id"/>
 3
 4  <!-- insert a cell for each recorded score -->
 5  <xsl:for-each select="/gradebook/task-list/task[@recorded='yes']">
 6      <xsl:sort select="@date" order="descending"/>
 7      <xsl:variable name="taskID" select="@id"/>
 8      <xsl:call-template name="insert_score">
 9          <xsl:with-param name="n"
```

```
10               select="key('student-index',►
   $id)/result[@ref=$taskID]/@score"/>
11        </xsl:call-template>
12    </xsl:for-each>
13    </table:table-row>
14 </xsl:template>
```

Line 2
: Our current context is a `<student>` node, so this sets the variable named `id` to the value of the student’s `id` attribute.

Lines 5-6
: We have to produce the student’s scores for each recorded task, so, as we have done before, we go to each recorded `<task>` element in turn, in reverse chronological order. At this point, our context has changed—it is now a `<task>` node.

Line 7
: Still in the context of the `<task>`, we save *its* `id` attribute in the `taskID` variable. In other words `taskID` contains the ID of our current task.

Lines 8-11
: The `insert_score` template takes a parameter named `n`, which is set to the current student’s score on the task being processed. The hardest part of this is line 10, so let us examine it in detail:

  `key('student-index', $id)`
  Switch back to the context of the current student by retrieving her node from the index table that we built at the beginning of the document. Our context is once again the student node.

  `result[@ref=$taskID]`
  For this student, find the `<result>` node whose `ref` attribute is the same as the ID of the task we are processing. (This will give us only one node, since task IDs are unique.) Our context is now that `<result>` element.

  `@score`
  Retrieve the `score` attribute from the result.

That takes care of the major templates. Now we write the named templates (subroutines) that we referred to earlier. First is the `insert-score` template, which creates a table cell whose value is the passed parameter `n`. If a null value has been passed (in case someone does not have a score for a task), then we insert an empty cell.

```
<xsl:template name="insert_score">
    <xsl:param name="n"/>

    <xsl:choose>
    <xsl:when test="$n != ''">
        <table:table-cell office:value-type="float"
```

```
          office:value="{$n}">
            <text:p><xsl:value-of select="$n"/></text:p>
        </table:table-cell>
    </xsl:when>
    <xsl:otherwise>
        <table:table-cell><text:p/></table:table-cell>
    </xsl:otherwise>
    </xsl:choose>
</xsl:template>
```

Finally, we have the template to figure out the column letter corresponding to the last column for each row. The pseudo-code is as follows:

- If the column number is less than or equal to 26, then the column name is a single letter. Extract it from the string `ABCDEFGHIJKLMNOPQRSTUVWXYZ`
- Otherwise, you have a two-letter column name. The first letter will be the column number divided by 26 and the second letter will be the column number modulo 26.

```
<xsl:template name="generate-col">
    <xsl:param name="n"/>
    <xsl:variable
       name="letters">ABCDEFGHIJKLMNOPQRSTUVWXYZ</xsl:variable>
    <xsl:choose>
        <xsl:when test="$n &lt;= 26">
            <xsl:value-of select="substring($letters, $n, 1)"/>
        </xsl:when>
        <xsl:otherwise>
            <xsl:value-of select="concat(
                substring($letters, floor(($n - 1) div 26) + 1, 1),
                substring($letters, (($n - 1) mod 26) + 1, 1))"/>
        </xsl:otherwise>
    </xsl:choose>
</xsl:template>
```

The actual code for two-letter columns has to subtract one from the column number before dividing (so that the math works out right), and must add one to the result, because characters are numbered starting at one.

Chapter 6. Drawings

Let's take a break from pure text and numbers to investigate OpenDocument's graphic elements. Before we get to the actual objects that you can draw on a page, we have to discuss the general format of the file.

A Drawing's styles.xml File

An `<office:master-styles>` element follows the `<office:automatic-styles>` in the `styles.xml` file. It defines the layers of a drawing and the master page. Example 6.1, "Master Styles in a Draw Document" shows this element:

Example 6.1. Master Styles in a Draw Document

```
  <draw:layer-set>
    <draw:layer draw:name="layout"/>
    <draw:layer draw:name="background"/>
    <draw:layer draw:name="backgroundobjects"/>
    <draw:layer draw:name="controls"/>
    <draw:layer draw:name="measurelines"/>
</draw:layer-set>

<style:master-page style:name="Default"
    style:page-master-name="PM1" draw:style-name="dp1"/>
```

These layers, unlike layers in most drawing programs, do not determine the stacking order of objects. They are just separate workspaces for holding various parts of the drawing. The `layout` layer contains most of your ordinary graphics. The `controls` layer contains buttons with actions attached to them (we won't examine those in this chapter). The `measurelines` layer contains lines which automatically calculate and display linear dimensions. The `background` and `backgroundobjects` layers are used in presentation documents, which we will cover in Chapter 7, Presentation.

A Drawing's content.xml File

OpenDocument drawing documents have an `<office:drawing>` element as the first child of the `<office:body>`. The `<office:automatic-styles>` element will start with an element like this:

```
<style:style style:name="dp1" style:family="drawing-page"/>
```

This is followed by all the styles to be applied to the document's graphic objects. Font specification is handled differently in a drawing file than it is in a word processing file. Instead of `office:font-face-decls` in the `styles.xml` file, font specifications are attached to the styles in the `content.xml` file. Instead of using `svg:font-family`, a drawing uses `fo:font-family`. Example 6.2, “Font Specification in a Drawing” shows a style for a paragraph in a drawing.

Example 6.2. Font Specification in a Drawing

```
<style:style style:name="P3" style:family="paragraph">
    <style:paragraph-properties
        fo:margin-left="0cm" fo:margin-right="0cm"
        fo:text-indent="0cm"/>
    <style:text-properties
        fo:font-family="Helvetica"
        style:font-family-generic="swiss"
        style:font-pitch="variable"/>
</style:style>
```

Following the style is the `<office:body>`, with an `<office:drawing>` element as its only child. The `<office:drawing>`’s first child is a `<draw:page>` element like this:

```
<draw:page draw:name="page1" draw:style-name="dp1"
    draw:master-page-name="Default">
```

This element contains the specifications for all the graphic objects drawn on the page.

Lines

Although the line is the simplest of the geometric figures, this explanation will be fairly lengthy, because we are defining our terms. Example 6.3, “Simple Horizontal Line” shows the `<draw:line>` elements for a horizontal line. The `draw:style-name` links this line with its style. The `draw:layer` attribute tells which workspace this line belongs to. The `svg:x1`, `svg:y1`, `svg:x2`, and `svg:y2` attributes define the beginning and ending points of the line as *length* specifiers. The `<draw:text-style-name>` attribute isn’t used in this example.

Example 6.3. Simple Horizontal Line

```
<draw:line draw:style-name="gr1"
    draw:layer="layout"
    svg:x1="1.905cm" svg:y1="1.429cm"
    svg:x2="9.155cm" svg:y2="1.429cm"
    draw:text-style-name="P1"/>
```

That’s all you need to define any sort of ordinary line; the different types of lines (arrows, colors, line style and width) are all handled in the style that is referenced by `draw:style-name`.

Line Attributes

Example 6.4, “Style for a Simple Line” shows the style for the preceding line. All it does is reference the default style and add two attributes that refer to text (which we will discuss in the section called “Attaching Text to a Line”).

Example 6.4. Style for a Simple Line

```
<style:style style:name="gr1" style:family="graphic"
  style:parent-style-name="standard">
    <style:graphic-properties
      draw:textarea-horizontal-align="center"
      draw:textarea-vertical-align="middle"/>
</style:style>
```

To change the thickness of a line, add an `svg:stroke-width` attribute to the `<style:graphic-properties>`. This attribute is given as a number followed by a *length* specification, such as `0.12cm`. The line’s color is controlled by the `svg:stroke-color`, given as a six-digit hexadecimal value such as `#ff0000`.

To make a dashed line, you add a `draw:stroke-dash` attribute; its value is the value of a `<draw:stroke-dash>` element which resides in the `styles.xml` file. Example 6.5, “Non-continuous Line” shows the XML for one of the standard dashed lines.

Example 6.5. Non-continuous Line

```
<draw:stroke-dash draw:name="Ultrafine_20_Dashed"
   draw:display-name="Ultrafine Dashed"
   draw:style="rect"
   draw:dots1="1" draw:dots1-length="0.051cm"
   draw:dots2="1" draw:dots2-length="0.051cm"
   draw:distance="0.051cm"/>
```

A dashed line is composed of alternating sequences of dots. `draw:dots1` and `draw:dots1-length` give the number of dots and the length of the first sequence.[7] `draw:dots2` and `draw:dots2-length` give the number of dots and length of the second sequence, and `draw:distance` gives the distance between the sequences. Finally, the `draw:style` tells whether the points of the dots and dashes are `rect` (square) or `round`, although, frankly, it’s hard to tell the difference.

Arrows

To add arrows to the beginning of a line, you use the `draw:marker-start` and `draw:marker-start-width` attributes. (A similar pair of `draw:marker-end` and `draw:marker-end-width` attributes define the arrow at the end of a line.)

[7] If there is no length for the first sequence of dots, then they appear as points of minimal size.

The `<draw:marker-start>` and `<draw:marker-end>` elements refer to a `<draw:marker>` element. The markers for the standard arrows from OpenOffice.org are in the `styles.xml` file, and the names are: `Arrow`, `Arrow_20_concave`, `Circle`, `Dimension_20_Lines`, `Double_20_Arrow`, `Line_20_Arrow`, `Rounded_20_large_20_Arrow`, `Rounded_20_short_20_Arrow`, `Small_20_Arrow`, `Square_20_45`, and `Symmetric_20_Arrow`.

Example 6.6, “Double Arrow Marker” shows the XML for the double arrow marker. If you don’t intend to create your own arrowhead definitions, you may skip this example and its following explanation.

Example 6.6. Double Arrow Marker

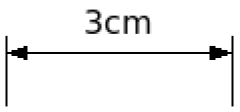
```
<draw:marker draw:name="Double_20_Arrow"
    svg:viewBox="0 0 1131 1918"
    svg:d="m737 1131 h394 l-564-1131 -567 1131 h398 l-398 787 ►
h1131 z"/>
```

To be *very* brief, the `svg:viewBox` attribute sets the coordinate system for the marker, and the `svg:d` attribute defines the outline of the marker, as it would be described in an SVG `<path>` element. We’ll discuss this attribute further in the section called “Polylines, Polygons, and Free Form Curves”.

Measure Lines

OpenDocument lets you specify “measure lines” which automagically show the line’s width, as in Figure 6.1, “Measure Line for a Horizontal Line”

Figure 6.1. Measure Line for a Horizontal Line



Example 6.7, “XML for a Measure Line” shows the XML for the measure line in Figure 6.1, “Measure Line for a Horizontal Line”. Note that the measure line belongs to the `measurelines` layer, not the normal `layout` layer.

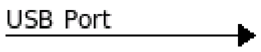
Example 6.7. XML for a Measure Line

```
<draw:measure draw:style-name="gr10"
  draw:text-style-name="P2"
  draw:layer="measurelines"
  svg:x1="10.901cm" svg:y1="4.656cm"
  svg:x2="13.901cm" svg:y2="4.656cm">
    <text:p text:style-name="P2">
        <text:measure text:kind="gap"/>
        <text:measure text:kind="value">3</text:measure>
        <text:measure text:kind="unit">cm</text:measure>
        <text:measure text:kind="gap"/>
    </text:p>
</draw:measure>
```

Attaching Text to a Line

There are times you want text to be “attached” to the line as a label so that moving the line moves the text along with it. Figure 6.2, “Arrow Labelled with Text” shows just such an arrow.[8]

Figure 6.2. Arrow Labelled with Text



Example 6.8, “XML for Labelled Arrow” shows the XML for the preceding figure.

Example 6.8. XML for Labelled Arrow

```
<draw:line draw:style-name="gr11"
  draw:text-style-name="P4"
  draw:layer="layout"
  svg:x1="2.858cm" svg:y1="1.852cm"
  svg:x2="6.165cm" svg:y2="1.852cm">
    <text:p text:style-name="P3">
        <text:span text:style-name="T1">USB Port</text:span>
    </text:p>
</draw:line>
```

As you may have suspected, the styles contain all the important information. Example 6.9, “Styles for Labelled Line” shows all the styles referenced in the preceding example.

Example 6.9. Styles for Labelled Line

```
<style:style style:name="gr11" style:family="graphic"
  style:parent-style-name="standard">
  <style:graphic-properties draw:marker-end="Symmetric_20_Arrow"
    draw:textarea-horizontal-align="left" ❶
    draw:textarea-vertical-align="bottom"
    fo:padding-top="0cm"
    fo:padding-bottom="0.1cm"
    fo:padding-left="0cm"/>
</style:style>

<style:style style:name="P4" style:family="paragraph"> ❷
    <style:paragraph-properties fo:text-align="start"/>
    <style:text-properties
      fo:font-family="&apos;Bitstream Vera Sans&apos;"
      style:font-pitch="variable" fo:font-size="9pt"/>
</style:style>
```

[8] It is possible to achieve the same effect in a more difficult way: by grouping a separate text and line object.

```
<style:style style:name="P3" style:family="paragraph">
    <style:paragraph-properties
      fo:margin-left="0cm" fo:margin-right="0cm"
      fo:text-align="start" fo:text-indent="0cm"/>
    <style:text-properties
      fo:font-family="&apos;Bitstream Vera Sans&apos;"
      style:font-pitch="variable" fo:font-size="9pt"/>
</style:style>

<style:style style:name="T1" style:family="text"> ❸
    <style:text-properties
      fo:font-family="&apos;Bitstream Vera Sans&apos;"
      style:font-pitch="variable" fo:font-size="9pt"/>
</style:style>
```

❶ `draw:textarea-horizontal-align` and `draw:textarea-vertical-align` place the text in relation to the area occupied by the line or arrow. The horizontal align can have values `left`, `center`, and `right`. The vertical align can have values `top`, `middle`, and `bottom`. The `fo:padding-`*`side`* attributes position the text relative to the line.

❷ This style combines the attributes of the following two styles; if you are creating a document programmatically, you don't have to create this style or reference it in the `<draw:line>` element.

❸ This style *is* necessary to set the font size properly.

Basic Shapes

Let's move on to two-dimensional objects: rectangles and squares, ellipses, circles, and arcs. Before looking at the individual shapes, it is necessary to discuss the style information they share.

Fill Styles

Two-dimensional objects have an outline and an interior, so their styles contain information about the lines that make up the outline and about the method used to fill the object's interior.

The styles for the outline are exactly the ones described in the section called "Line Attributes". The interior style is specified with the `draw:fill` attribute, which may have a value of `none`, `solid`, `gradient`, `hatch`, or `bitmap`. If you want an unfilled object, you *must* specify `none`. If you leave off this attribute, the object will be filled with the default color (which is set in the `<style:style style:name="standard" style:famliy="graphic">` element in `styles.xml`).

Solid Fill

Solid fills are the easiest; in addition to the specification of `draw:fill="solid"`, you specify the `draw:fill-color` attribute as a six-digit hexadecimal value. Example 6.10, “Solid Fill Style” shows the XML for a style which will draw objects with a 0.1 centimeter red border and a solid light blue fill.

Example 6.10. Solid Fill Style

```
<style:style style:name="gr13"
  style:family="graphic" style:parent-style-name="standard">
    <style:graphic-properties draw:stroke="solid"
        svg:stroke-width="0.1cm"
        svg:stroke-color="#ff0000"
        draw:fill="solid" draw:fill-color="#e6e6ff"/>
</style:style>
```

Gradient Fill

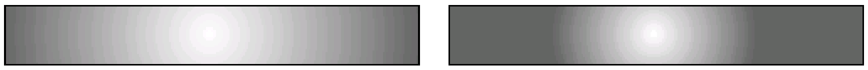
To fill an area with a gradient, set the `draw:fill` attribute to `gradient`, and set the `draw:fill-gradient-name` attribute to refer to a `<draw:gradient>` element in the `styles.xml` file.[9]

The `<draw:gradient>` element will have the corresponding `draw:name` attribute (with hex characters encoded as `_xx_`), a `draw:display-name`, and a `draw:style` attribute that specifies the gradient type: `linear` (from edge to edge), `axial` (from center outwards), `radial` (circular), `ellipsoid`, `square`, and `rectangular`.

All gradients have a `draw:start-color` and `draw:end-color` attribute. You may specify their intensity with the `draw:start-intensity` and `draw:end-intensity` attributes, each of which has values ranging from `0%` to `100%`; the default is 100%.

All gradients also have a `draw:border` attribute, again expressed as a percentage, which is used to scale a border which is filled by the start or end color only. The default is 0%. Figure 6.3, “Effect of Border on Gradients” shows two radial gradient fills. The gradient on the left has no border; the gradient on the right has a border of 50%.

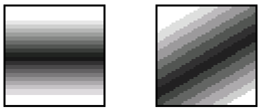
Figure 6.3. Effect of Border on Gradients



[9] If you are creating a file on your own, you may put gradients in the `content.xml` file.

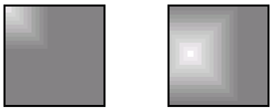
All gradients may specify a `draw:angle` attribute that rotates the gradient. Figure 6.4, "Effect of Angle on Gradients" shows two axial gradients; the one on the left has an angle of zero, and the one on the right has an angle of 30 degrees. Angles are measured counterclockwise from 0 degrees, which is at the "three o'clock" position. The value is expressed in tenths of a degree, so the value for the rotated gradient would be `300`. The default value is zero. In the case of a radial gradient, this attribute has no effect, since circles look alike at every angle.

Figure 6.4. Effect of Angle on Gradients



Finally, for all gradients *except* linear and axial, you may specify the center *x* and *y* coordinate where the gradient starts. The `draw:cx` and `draw:cy` attributes have values ranging from `0%` to `100%`. Figure 6.5, "Effect of Center Coordinates on Gradients" shows a rectangular gradient with a center *x* and *y* of 0% on the left, and a center *x* of 25% and center *y* of 50% on the right.

Figure 6.5. Effect of Center Coordinates on Gradients



Example 6.11, "XML Representation of Gradients" shows the XML for the gradients in the preceding figures. The intensity attributes have been removed to save space.

Example 6.11. XML Representation of Gradients

```
<!-- radial gradient, light gray to white, centered, 0% border -->
<draw:gradient draw:name="radial_20_borderless"
    draw:display-name="radial borderless"
    draw:style="radial"
    draw:cx="50%" draw:cy="50%"
    draw:start-color="#999999" draw:end-color="#ffffff"
    draw:border="0%"/>

 <!-- radial gradient, gray to white, centered, 50% border -->
 <draw:gradient draw:name="radial_20_bordered"
    draw:display-name="radial bordered"
    draw:style="radial"
    draw:cx="50%" draw:cy="50%"
    draw:start-color="#666666" draw:end-color="#ffffff"
    draw:start-intensity="100%" draw:end-intensity="100%"
draw:border="50%"/
```

```
<!-- axial gradient, black to white, 30% border -->
<draw:gradient draw:name="axial_20_black_20_to_20_white"
    draw:display-name="axial black to white"
    draw:style="axial"
    draw:start-color="#000000" draw:end-color="#ffffff"
    draw:angle="0" draw:border="30%"/>

<!-- axial gradient, black to white, 30% border, rotated 30 degrees
-->
<draw:gradient
    draw:name="axial_20_30_20_degree_20_black_20_to_20_white"
    draw:display-name="axial 30 degree black to white"
    draw:style="axial"
    draw:start-color="#000000" draw:end-color="#ffffff"
    draw:angle="300" draw:border="30%"/>

<!-- rectangular gradient, gray to white, centered at top left -->
<draw:gradient draw:name="rectangular_20_0_20_0"
    draw:display-name="rectangular 0 0"
    draw:style="rectangular"
    draw:cx="0%" draw:cy="0%"
    draw:start-color="#808080" draw:end-color="#ffffff"
    draw:angle="0" draw:border="0%"/>

<!-- rectangular gradient, gray to white, centered at (25%, 50%) -->
<draw:gradient draw:name="rectangular_20_25_20_50"
    draw:display-name="rectangular 25 50"
    draw:style="rectangular"
    draw:cx="25%" draw:cy="50%"
    draw:start-color="#808080" draw:end-color="#ffffff"
    draw:angle="0" draw:border="0%"/>
```

Hatch Fill

Compared to gradients, hatch fills are simplicity itself. To fill an object with a hatch, set the object's `draw:fill` attribute to `hatch`, and set the `draw:fill-hatch-name` attribute to refer to a `<draw:hatch>` element in the `styles.xml` file.

Within a `<draw:hatch>` element, specify these attributes in addition to the `draw:name` and `draw:display-name`.

`draw:style`
: Number of lines in the hatch: `single`, `double` (lines at right angles), or `triple` (lines at right angles, with one diagonal).

`draw:color`
: The hatching color, as a six-digit hex value.

`draw:distance`
: The spacing between hatch lines, in a form such as `0.18cm`.

`draw:rotation`
: A number in tenths of a degree; thus a 45 degree rotation has a value of `450`.

Example 6.12, “XML for Hatch Fill” shows the XML for a widely-spaced, deep red, single-line hatch at a 45 degree angle.

Example 6.12. XML for Hatch Fill

```
<draw:hatch draw:name="Red_20_Diagonal"
    draw:display-name="Red Diagonal"
    draw:style="single"
    draw:color="#800000"
    draw:distance="0.18cm"
    draw:rotation="450"/>
```

If you want a background color to fill the spaces between the hatch lines, then set the object’s `draw:fill-color` as well as its `draw:fill-hatch` attribute.

Bitmap Fill

To fill an object with a bitmap image, set the object’s `draw:fill` attribute to `bitmap`, and set the `draw:fill-image-name` attribute to refer to a `<draw:fill-image>` element in the `styles.xml` file.

Within a `<draw:fill-image>` element, you have the standard `draw:name` attribute and `draw:display-name` plus:

`xlink:href`
: A reference to the image file; it begins with `#Pictures/` and is followed by the internal file name. The file is stored in the `Pictures` directory.

`xlink:type`, `xlink:show`, `xlink:actuate`
: These three items always have the values `simple` (for `xlink:type`), `embed` (for `xlink:show`), and `onLoad` (for `xlink:actuate`).

Example 6.13, “XML for Bitmap Fill” shows the XML for a bitmap fill.

Example 6.13. XML for Bitmap Fill

```
<draw:fill-image draw:name="Wall"
    xlink:href="#Pictures/100000000000000B4000000874138D207.png"
    xlink:type="simple" xlink:show="embed" xlink:actuate="onLoad"/>
```

Drop Shadows

You may give any object a drop shadow by specifying the following attributes:

`draw:shadow`
: The value should be set to `visible`

`draw:shadow-color`
: The shadow color, expressed as a six-digit hex value.

`draw:shadow-opacity`
: The value ranges from `0%` (transparent) to `100%` (completely opaque).

`draw:shadow-offset-x`, `draw:shadow-offset-y`
: The distance from the upper left corner of the object to the upper left corner of the shadow, specified in centimeters. Negative numbers are left and upwards of the object.

Rectangles

Now that we know how to outline and fill an object, let's find out how to draw objects. The simplest object of all is a rectangle, expressed with the `<draw:rect>` element. You define a rectangle by the coordinates of its upper left corner, (`svg:x` and `svg:y`) and its width and height (`svg:width` and `svg:height`). As with lines, rectangles will have a `draw:layer` of `layout`. Finally, a `draw:style-name` attribute will define the fill and outline style for the rectangle. You can see all of this put together in Example 6.14, "XML for a Rectangle".[10]

Example 6.14. XML for a Rectangle

```
 <draw:rect draw:layer="layout"
    svg:x="2.105cm" svg:y="2.225cm"
    svg:width="2.827cm" svg:height="1.203cm"
    draw:style-name="gr1"/>
```

You may draw a rounded rectangle by specifying a `draw:corner-radius` attribute with a *length* as its value, such as `0.5cm`.

Circles and Ellipses

To draw a circle or an ellipse, use a `<draw:circle>` or `<draw:ellipse>` element. The attributes of this tag are exactly the same as those for a rectangle. Instead of drawing a rectangle, the `svg:x`, `svg:y`, `svg:width`, and `svg:height` define the *bounding box* of the circle or ellipse. In other words, the circle or ellipse is inscribed in the rectangle that you have specified. Example 6.15, "XML for Circle and Ellipse" shows the XML for a circle and an ellipse.

Example 6.15. XML for Circle and Ellipse

```
<draw:circle draw:style-name="gr2"
    draw:text-style-name="P1" draw:layer="layout"
    svg:x="5.533cm" svg:y="1.984cm"
    svg:width="1.745cm" svg:height="1.745cm"/>

<draw:ellipse draw:style-name="gr1"
    draw:text-style-name="P1" draw:layer="layout"
    svg:x="1.504cm" svg:y="2.044cm"
    svg:width="2.466cm" svg:height="1.445cm"/>
```

[10] All objects can also have a `draw:text-style-name` attribute if you attach text to them, as described in the section called "Attaching Text to a Line", but that attribute is not included in these examples.

Arcs and Segments

You may also draw elliptical and circular arcs and segments. For all of these, you must specify the `draw:start-angle` and `draw:end-angle` attributes in degrees. The `draw:kind` attribute tells what part of the ellipse you are drawing: `section` for a pie-shaped slice, `cut` for a chord cut across the ellipse or circle, and `arc` for an unfilled arc. Example 6.16, “XML for Arcs and Segments” shows an example of each of these.

Example 6.16. XML for Arcs and Segments

```
<!-- elliptical pie slice -->
<draw:ellipse draw:style-name="gr2"
    draw:layer="layout"
    svg:width="1.932cm" svg:height="1.482cm"
    svg:x="4.842cm" svg:y="2.275cm"
    draw:kind="section"
    draw:start-angle="319.5" draw:end-angle="71.09/">

<!-- circle segment -->
<draw:circle draw:style-name="gr3"
    draw:layer="layout"
    svg:width="1.587cm" svg:height="1.587cm"
    svg:x="7.938cm" svg:y="1.218cm"
    draw:kind="cut"
    draw:start-angle="255.47" draw:end-angle="40.23"/>

<!-- elliptical arc -->
<draw:ellipse draw:style-name="gr1"
    draw:layer="layout"
    svg:width="1.958cm" svg:height="1.402cm"
    svg:x="2.117cm" svg:y="1.588cm"
    draw:kind="arc"
    draw:start-angle="360" draw:end-angle="70.25"/>
```

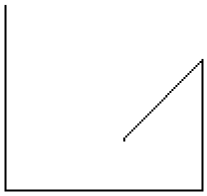
Polylines, Polygons, and Free Form Curves

If you want anything other than a basic shape, you have to use a polyline, polygon, or free-form curve. For all of these, you must establish a *bounding box* and a *coordinate system*. The bounding box is a rectangle which entirely contains the object, and is specified with `svg:x`, `svg:y`, `svg:width`, and `svg:height` attributes. all measured in centimeters.

The coordinate system is established with the `svg:viewBox` attribute which has four numbers: the minimum *x*-coordinate, minimum *x*-coordinate, width, and height of the coordinate system. In OpenDocument, the minimum coordinates are zero, and the width and height are set to the width and height of the bounding box times 1000. Thus, a coordinate of (250, 1602) will be 0.25 centimeters from the left corner of the bounding box and 1.602 centimeters from the top of the bounding box.

A *polyline* is simply a series of connected straight lines. Their coordinates are described as pairs of *x*,*y* coordinates separated by whitespace. The polyline shown in Figure 6.6, “Simple Polyline” has the XML representation in Example 6.17, “XML for a Polyline”.

Figure 6.6. Simple Polyline



Example 6.17. XML for a Polyline

```
<draw:polyline draw:style-name="gr1" draw:layer="layout"
    svg:x="8.335cm" svg:y="4.948cm"
    svg:width="2.593cm" svg:height="2.434cm"
    svg:viewBox="0 0 2594 2435"
    draw:points="0,0 0,2434 2593,2434 2593,714 1561,1746 1561,1773"/>
```

Polygons use the `<draw:polygon>` element, which has exactly the same attributes as `<draw:polyline>`. You do not have to close the polygon by repeating the first coordinate at the end of the `draw:points` attribute; the application will automatically close the figure for you when it draws the polygon.

OpenOffice.org’s Coordinate System

OpenOffice.org does not allow you to set an arbitrary coordinate system; it sets the coordinate system to one thousand units per centimeter. All *x*- and *y*- positions, widths, and heights are measured in centimeters, independent of the “unit of measurement” setting.

For all other curves (Bézier curves and freeform drawing), OpenDocument uses a `<draw:path>` element, which is modeled after the SVG `<path>` element. The `svg:d` attribute describes the path data, and has a value consisting of command letters and coordinates. For example, the `M` command moves the pen to the given *x*- and *y*-coordinates. `L` draws a line to the given coordinates, and `Z` is the command for closing a polygon. Thus, the path data for a trapezoid would look like this:

```
svg:d="M 300 0  L 700 0 L 1000 1000 L 0 1000 Z"
```

Lowercase letters take relative coordinates (relative to the last point drawn). Thus, the preceding trapezoid, expressed in relative coordinates, is:

```
svg:d="m 300 0  l 400 0 l 300 1000 l -1000 0 z"
```

In the preceding example, the first coordinate is relative to (0,0), the upper left of the bounding box. If you eliminate duplicated commands and unnecessary whitespace, the preceding examples can be reduced to:[11]

```
svg:d="M300 0L700 0 1000 1000 0 1000Z"
svg:d="m300 0l400 0 300 1000-1000 0z"
```

Table 6.1, "Path Commands" summarizes the path commands available in OpenDocument. This table is taken from *SVG Essentials*, published by O'Reilly & Associates. When creating paths, uppercase commands use absolute coordinates and lowercase commands use relative coordinates. Some applications may support only a subset of these commands.

Table 6.1. Path Commands

Command	Arguments	Effect
`M` `m`	*`x y`*	Move to given coordinates.
`L` `l`	*`x y`*	Draw a line to the given coordinates. You may supply multiple sets of coordinates to draw a polyline.
`H` `h`	*`x`*	Draw a horizontal line to the given *x*-coordinate.
`V` `v`	*`y`*	Draw a vertical line to the given *y*-coordinate.
`A` `a`	*`rx ry x-axis-rotation large-arc sweep x y`*	Draw an elliptical arc from the current point to (*x*, *y*). The points are on an ellipse with x-radius *`rx`* and y-radius *`ry`*. The ellipse is rotated *`x-axis-rotation`* degrees. If the arc is less than 180 degrees, *`large-arc`* is zero; if greater than 180 degrees, *`large-arc`* is one. If the arc is to be drawn in the positive direction, *`sweep`* is one; otherwise it is zero.
`Q` `q`	*`x1 y1 x y`*	Draw a quadratic Bézier curve from the current point to (*x*, *y*) using control point (*x1*, *y1*).
`T` `t`	*`x y`*	Draw a quadratic Bézier curve from the current point to (*x*, *y*). The control point will be the reflection of the previous `Q` command's control point. If there is no previous curve, the current point will be used as the control point.
`C` `c`	*`x1 y1 x2 y2 x y`*	Draw a cubic Bézier curve from the current point to (*x*, *y*) using control point (*x1*, *y1*) as the control point for the beginning of the curve and (*x2*, *y2*) as the control point for the endpoint of the curve.
`S` `s`	*`x2 y2 x y`*	Draw a cubic Bézier curve from the current point to (*x*, *y*), using (*x2*, *y2*) as the control point for this new endpoint. The first control point will be the reflection of the previous `C` command's ending control point. If there is no previous curve, the current point will be used as the first control point.

[11] The first `L` is also unnecessary; points after a first Move are presumed to be lines.

Warning

OpenOffice.org does not support the arc path commands `A` and `a`.

Adding Text to Drawings

Text in a graphic is contained in a `<draw:frame>` element. the drawing style. As with other objects, it has a `draw:layer` of `layout` and sets the location and dimensions with `svg:x`, `svg:y`, `svg:width`, and `svg:height`. The `<draw:frame>` also contains a `draw:style-name` attribute that refers to a `style:style` element in the `content.xml` file. This `<style:style>` contains a `<style:graphic-properties>` element that sets

- `draw:stroke`
- `svg:stroke-color`
- `draw:fill`
- `draw:fill-color`
- `draw:textarea-vertical-align` (`top`, `middle`, or `bottom`)
- `draw:textarea-horizontal-align` (`left`, `right`, `center`, or `justify`)
- `draw:writing-mode` (set to `tb-rl` for vertical text; omit it for horizontal text)

You may specify how the text area grows to fit the text with the `draw:auto-grow-height` and `draw:auto-grow-width` attributes, which take values of `true` and `false`.

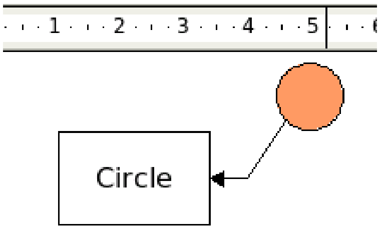
Within the `<draw:frame>` is a `<draw:text-box>` element, which contains one or more `<text:p>` elements that describe the actual text. Example 6.18, “Simple Text Box” shows the XML fragments for a simple text area that displays the words `Hello, text.` on a peach-colored background with a purple border.

Example 6.18. Simple Text Box

```
<!-- within office:automatic-styles section -->
<style:style style:name="gr1" style:family="graphic"
  style:parent-style-name="standard">
    <style:graphic-properties
        svg:stroke-width="0.051cm" svg:stroke-color="#800080"
        draw:marker-start-width="0.381cm"
        draw:marker-end-width="0.381cm"
        draw:fill="solid" draw:fill-color="#ffcc99"
        draw:textarea-horizontal-align="left"
        draw:auto-grow-height="true" draw:auto-grow-width="true"
        fo:min-height="0cm" fo:min-width="0cm"
        fo:padding-top="0.15cm" fo:padding-bottom="0.15cm"
        fo:padding-left="0.275cm" fo:padding-right="0.275cm"/>
</style:style>

<!-- within document body -->
<draw:frame draw:style-name="gr1" draw:text-style-name="P1"
  draw:layer="layout"
  svg:x="2.3cm" svg:y="2.25cm"
  svg:width="4.05cm" svg:height="1.04cm">
    <draw:text-box>
        <text:p text:style-name="P1">Hello, text.</text:p>
    </draw:text-box>
</draw:frame>
```

A callout is implemented via a `<draw:caption>` element. In addition to all the attributes of the `draw:frame` element, it specifies the endpoint of the callout line with the `draw:caption-point-x` and `draw:caption-point-y` attributes. Their values are relative to the upper left corner of the callout's bounding box. Example 6.19, "XML for Callout Text" shows the style and body XML for the callout depicted in Figure 6.7, "Callout Text in a Drawing". (Just the callout, not the circle.) The arrow is considered to be the `draw:marker-end`.

Figure 6.7. Callout Text in a Drawing



Example 6.19. XML for Callout Text

```
<!-- style information -->
<style:style style:name="gr2" style:family="graphic"
  style:parent-style-name="standard">
    <style:graphic-properties
        draw:marker-start=""
        draw:marker-end="Symmetric_20_Arrow"
        draw:fill="none" draw:fill-color="#ffffff"
        draw:textarea-horizontal-align="justify"
        draw:textarea-vertical-align="middle"
        draw:auto-grow-height="true" draw:auto-grow-width="true"
        fo:min-height="1.238cm" fo:min-width="2.128cm"
        fo:padding-top="0.1cm" fo:padding-bottom="0.1cm"
        fo:padding-left="0.1cm" fo:padding-right="0.1cm"/>
</style:style>

<!-- body information -->
<draw:caption draw:style-name="gr2" draw:text-style-name="P3"
  draw:layer="layout"
  svg:x="1.907cm" svg:y="2.594cm"
  svg:width="2.328cm" svg:height="1.438cm"
  draw:caption-point-x="3.491cm" draw:caption-point-y="-0.158cm">
    <text:p text:style-name="P3">
        <text:span text:style-name="T1">Circle</text:span>
    </text:p>
</draw:caption>
```

Rotation of Objects

A rotated object in an OpenDocument drawing has a `draw:transform` attribute in the element that describes that object. The value of this attribute is a whitespace-separated list of mathematical functions to be performed on the object. These transformations include:

`rotate( )`
: The argument to this function is the number of *radians* to rotate, not degrees, as in SVG. To convert from degrees to radians, multiply by pi (3.141592653...) and divide by 180. The object is rotated about its pivot point, which is normally the object's geometric center. Angles are measured with zero at the "three o'clock" position, increasing in a counterclockwise direction.

`translate( )`
: This function moves the object to the specified location, given as two *lengths* separated by whitespace, as in `translate(1cm 1.3cm)`.

`skewX( ), skewY( )`
: These functions take a single argument, given in radians, which tells how to slant the object in the *x*-direction or *y*-direction.

`scale( )`
: This function takes two arguments: the factor by which to scale all *x*-coordinates and the factor by which to scale all *y*-coordinates. If you give only one argument, then both *x*- and *y*-coordinates will be scaled by this factor.

It is possible to have a series of transformations separated by whitespace. The order of transformations is important. When you rotate an object in a drawing, the application will often generate a rotate followed by a translate in order to position the object properly.

Case Study: Weather Diagram

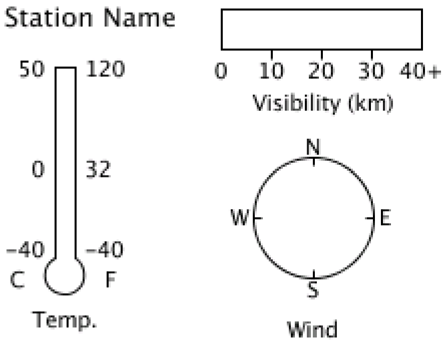
We now have enough information to create a drawing from scratch. We will use XSLT to create an OpenDocument drawing from data in the Weather Observation Markup Format (OMF), defined at `http://zowie.metnet.navy.mil/~spawar/JMV-TNG/XML/OMF.html`. OMF is, for the most part, a wrapper for several different types of weather reports. The OMF elements add annotation, decoded information, and quantities calculated from the raw reports. Here is a sample report:

```
<Reports TStamp="997568716">
<SYN Title='AAXX' TStamp='997573600' LatLon='37.567, 126.967'
BId='471080'
SName='RKSL, SEOUL' Elev='86'>
<SYID>47108</SYID>
```

```
<SYG T='22.5' TD='14.1' P='1004.1' P0='1014.1' Pd='0 0.1' Vis='22000'
Ceiling='INF' Wind='30-70, 1.5' WX='NOSIG' Prec=' ' Clouds='44070'>
32972 40703 10225 20141 30041 40141 50001 84070
</SYG></SYN>
</Reports>
```

Our objective is to extract the reporting station, the date and time, temperature, wind speed and direction, and visibility from the report. These data will be filled into the graphic template of Figure 6.8, "Graphic Weather Template".

Figure 6.8. Graphic Weather Template



The OMF format attributes that we're interested in are listed here, along with the plan for displaying them in the final graphic. The first two required attributes come from the `<SYN>` element, the rest are optional attributes from its child `<SYG>` element.

`SName`
: The reporting station's call letters, possibly followed by a comma and the station's full name. The final graphic will represent this as text.

`T`
: The air temperature in degrees Celsius. This will be displayed by coloring in the thermometer to the appropriate level. If the temperature is greater than zero, then the coloring will be red; if less than or equal to zero, then it will be blue.

`Wind`
: A direction and speed, separated by a comma. The direction is measured in degrees; 0 indicates wind blowing from true north, and 270 indicates wind from the west. This will be represented by a line on the compass.

  Wind direction may also be expressed as two numbers separated by a dash, indicating a range of directions. Thus, `0-40` indicates wind from the north to north-east. In this case, two dashed lines will be drawn on the compass.

The wind speed is expressed in meters per second. If two numbers are given, separated by a dash, the second number indicates the speed of gusts of wind. This information will be displayed in text form.

`Vis`

Surface visibility in meters, or the value `INF` for unlimited visibility. The final graphic will represent this by filling in a horizontal bar. Any visibility above forty kilometers will be presumed to be unlimited.

Styles for the Weather Drawing

Let us first examine the styles we will need for this drawing:

- A solid line. We will use this line for the outlines of the thermometer, wind, and visibility meters. It will also be used for the direction line of the wind.
- A dashed line. We will use this if there is more than one direction line for the wind.
- A red fill with no outline, used for temperatures greater than zero.
- A blue fill with no outline, used for temperatures less than or equal to zero.
- A green fill with no outline, used to fill the visibility meter.
- A graphics style for text with no outline and no fill
- Ten-point text styles for the small text; left-aligned, centered, and right-aligned
- A twelve-point text style for the larger text.

This gives us enough to start our stylesheet. [The full transformation is file `weather.xsl` in directory `ch06` in the downloadable example files.] We start out with the root element and its namespace declarations.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns:office="urn:oasis:names:tc:opendocument:xmlns:office:1.0"
    xmlns:text="urn:oasis:names:tc:opendocument:xmlns:text:1.0"
    xmlns:table="urn:oasis:names:tc:opendocument:xmlns:table:1.0"
    xmlns:draw="urn:oasis:names:tc:opendocument:xmlns:drawing:1.0"
    xmlns:style="urn:oasis:names:tc:opendocument:xmlns:style:1.0"
    xmlns:fo="urn:oasis:names:tc:opendocument:xmlns:►
xsl-fo-compatible:1.0"
    xmlns:svg="urn:oasis:names:tc:opendocument:xmlns:►
svg-compatible:1.0"

    xmlns:label="urn:evc-cit.info:examples:internal-use-only"
>
```

Whoa! That last namespace isn't one that you have seen before. We're going to insert some custom elements of our own into this stylesheet to make it easier to put labels on the graphics. These elements will need a namespace, so we have made up a prefix (`label`) and a URI for them (`urn:evc-cit.info:examples:internal-use-only`). A URI just needs to be unique; it doesn't have to point to anything on the web—and in this case, we don't want it to.

Now, on to the rest of the setup:

```
<xsl:output method="xml" indent="yes"/>

<xsl:template match="/">
<office:document-content
    xmlns:office="urn:oasis:names:tc:opendocument:xmlns:office:1.0"
    xmlns:style="urn:oasis:names:tc:opendocument:xmlns:style:1.0"
    xmlns:text="urn:oasis:names:tc:opendocument:xmlns:text:1.0"
    xmlns:table="urn:oasis:names:tc:opendocument:xmlns:table:1.0"
    xmlns:draw="urn:oasis:names:tc:opendocument:xmlns:drawing:1.0"
    xmlns:fo="urn:oasis:names:tc:opendocument:xmlns:►
xsl-fo-compatible:1.0"
    xmlns:svg="urn:oasis:names:tc:opendocument:xmlns:►
svg-compatible:1.0"
    office:version="1.0">
<office:scripts/>

<office:automatic-styles>
    <style:style style:name="dp1" style:family="drawing-page"/>

    <style:style style:name="solidLine" style:family="graphic"
     style:parent-style-name="standard">
        <style:graphic-properties
            svg:stroke-color="#000000"
            draw:fill="none"/>
    </style:style>

    <draw:stroke-dash draw:name="3 by 3" draw:style="rect"
        draw:dots1="3" draw:dots1-length="0.203cm"
        draw:dots2="3" draw:dots2-length="0.203cm"
        draw:distance="0.203cm"/>

    <style:style style:name="dashLine" style:family="graphic"
     style:parent-style-name="standard">
        <style:graphic-properties
            draw:stroke="dash" draw:stroke-dash="3 by 3"
            svg:stroke-color="#000000"
            draw:fill="none"/>
    </style:style>

    <style:style style:name="redfill" style:family="graphic"
     style:parent-style-name="standard">
        <style:graphic-properties draw:stroke="none"
            draw:fill="solid" draw:fill-color="#ff0000"/>
    </style:style>

    <style:style style:name="bluefill" style:family="graphic"
     style:parent-style-name="standard">
        <style:graphic-properties draw:stroke="none"
            draw:fill="solid" draw:fill-color="#0000ff"/>
    </style:style>

    <style:style style:name="greenfill" style:family="graphic"
     style:parent-style-name="standard">
        <style:graphic-properties draw:stroke="none"
            draw:fill="solid" draw:fill-color="#008000"/>
    </style:style>

    <style:style style:name="textgraphics" style:family="graphic"
```

```
      style:parent-style-name="standard">
        <style:graphic-properties draw:stroke="none" draw:fill="none"
            draw:auto-grow-width="true" draw:auto-grow-height="true"/>
    </style:style>

    <style:style style:name="smallLeft" style:family="paragraph">
        <style:text-properties
            fo:font-size="10pt"
            fo:font-family="Bitstream Vera Sans"/>
    </style:style>

    <style:style style:name="smallRight" style:family="paragraph">
        <style:paragraph-properties fo:text-align="end"/>
        <style:text-properties
            fo:font-size="10pt"
            fo:font-family="Bitstream Vera Sans"/>
    </style:style>

    <style:style style:name="smallCenter" style:family="paragraph">
        <style:paragraph-properties fo:text-align="center"/>
        <style:text-properties
            fo:font-size="10pt"
            fo:font-family="Bitsream Vera Sans"/>
    </style:style>

    <style:style style:name="largetext" style:family="paragraph">
        <style:text-properties
            fo:font-size="12pt"
            fo:font-family="Bitstream Vera Sans"/>
    </style:style>

</office:automatic-styles>

<office:body>
    <office:drawing>
        <draw:page draw:name="page1" draw:style-name="dp1"
            draw:master-page-name="Default">
            <xsl:apply-templates select="Reports"/>
        </draw:page>
    </office:drawing>
</office:body>
</office:document-content>
</xsl:template>

<!-- other templates go here -->
</xsl:stylesheet>
```

Objects in the Weather Drawing

To make things modular (and to keep our sanity), we will handle the four objects in separate templates, from easiest to most difficult.

```
<xsl:template match="Reports">
    <xsl:apply-template select="SYN/@SName"/>
    <xsl:apply-template select="SYN/SYG/@Vis"/>
    <xsl:apply-template select="SYN/SYG/@Wind"/>
    <xsl:apply-template select="SYN/SYG/@T"/>
</xsl:template>
```

The Station Name

The station name is pure text which goes into a `<draw:text-box>` element (within a `<draw:frame>`). We don't want the station abbreviation, so we will use just the part that follows the comma and blank. We must set both the `svg:width` and `svg:height` to display the text properly, even though we have set `auto-grow-width` and `auto-grow-height` to `true` in the style. A little math tells us that one point is 0.0353 centimeters, so a height of one-half centimeter will easily accommodate a twelve-point font. To calculate the width, we will multiply the number of characters by twelve (the point size) and convert that to centimeters. This will be longer than we need, since not all characters will have maximum width. Here's the resulting template.

```
<xsl:template match="@SName">
    <xsl:variable name="shortName"
        select="substring-after(., ', ')"/>
    <xsl:variable name="nameWidth"
        select="(string-length($shortName) * 12) * 0.0353"/>

<draw:frame
    draw:style-name="textgraphics"
    draw:layer="layout"
    svg:x="0.75cm" svg:y="2cm"
    svg:width="{$nameWidth}cm" svg:height=".5cm">
    <draw:text-box>
    <text:p text:style-name="largetext"><xsl:value-of
        select="$shortName"/></text:p>
    </draw:text-box>
</draw:frame>
</xsl:template>
```

The Visibility Bar

On to the visibility bar. To make the math easy, we'll make the bar eight centimeters wide and one centimeter tall.

```
<xsl:template match="@Vis">

<!-- calculate the number of centimeters to fill -->
<xsl:variable name="visWidth"> ❶
   <xsl:choose>
       <xsl:when test=". = 'INF'">8</xsl:when>
       <xsl:when test=". &gt; 40000">8</xsl:when>
       <xsl:otherwise>
           <xsl:value-of select=". * 8 div 40000.00"/>
       </xsl:otherwise>
   </xsl:choose>
</xsl:variable>

<!-- draw the filled area --> ❷
<draw:rect draw:style-name="greenfill" draw:layer="layout"
   svg:x="6cm" svg:y="2cm"
   svg:width="{$visWidth}cm" svg:height="1cm" />
```

```
<!-- draw the empty bar -->
<draw:rect draw:style-name="solidLine" draw:layer="layout"
   svg:x="6cm" svg:y="2cm"
   svg:width="8cm" svg:height="1cm" />

<!-- tick marks --> ❸
<draw:path draw:style-name="solidLine" draw:layer="layout"
   svg:x="8cm" svg:y="3cm" svg:width="8cm" svg:height="1cm"
   svg:viewBox="0 0 8000 1000"
   svg:d="m 0 0 l 0 125 m 2000 -125 l 0 125 m 2000 -125 l 0 125"/>

<!-- create text under the tick marks --> ❹
<xsl:for-each select="document('')/*/label:visibility/label:text">
    <draw:frame draw:style-name="textgraphics" draw:layer="layout"
    svg:y="3.125cm"
    svg:width="0.8cm" svg:height=".5cm">
       <xsl:attribute name="svg:x">
           <xsl:value-of select="@x + 5.625"/>
           <xsl:text>cm</xsl:text>
       </xsl:attribute>
       <draw:text-box>
           <text:p text:style-name="smallCenter"><xsl:value-of
            select="."/></text:p>
       </draw:text-box>
    </draw:frame>
</xsl:for-each>

<draw:frame draw:style-name="textgraphics" ❺
    draw:layer="layout"
    svg:x="6cm" svg:y="3.7cm" svg:height="1cm" svg:width="8cm">
    <draw:text-box>
      <text:p text:style-name="smallCenter">Visibility (m)</text:p>
      <text:p text:style-name="smallCenter"><xsl:value-of select="."/>
      </text:p>
    </draw:text-box>
</draw:frame>

</xsl:template>

<label:visibility> ❻
    <label:text x="0">0</label:text>
    <label:text x="2">10</label:text>
    <label:text x="4">20</label:text>
    <label:text x="6">30</label:text>
    <label:text x="8">40+</label:text>
</label:visibility>
```

❶ To prevent the filled area from overflowing the boundary, we have to test to see if we have the special value of `INF` (unlimited visibility) or a value greater than 40,000. If so, fill all eight centimeters of the bar.

❷ If object A's XML comes before object B's XML in the file, then object A is behind object B. Thus, we draw the filled-in area first. This guarantees that it will not obscure the outline of the visibility bar.

❸ We start the *x*-coordinate at 8 centimeters, where the first tick mark lies. This is because OpenOffice.org became cranky when we set the *x*-coordinate to 10 centimeters and started the path with `m2000 0`, insisting upon moving it back to (0,0).

❹ Each of the numbers underneath the visibility bar requires a `<draw:frame>` element. Each of these elements has a large number of attributes, differing only in the `svg:x` value. Copy-and-paste will work, but if a modification of the drawing required a change to one of the other attributes, we would have to change all the elements.

What we will do is keep a table of the labels' text and their offset from the starting *x*-position, and then use XSLT to process that table. This table will be stored in the form of elements with the `label:` namespace.

In order to access this information, we use the XPath expression `document('')/*/label:visibility/label:text`. It will select all `<label:text>` elements within `<label:visiblity>` elements within this entire document (`document('')/*`).

❺ We finish the template with the words "Visibility (m)" and the visiblity value in a framed text box.

❻ Here are the labels and their offsets that the template uses. The offsets are in centimeters.

The Wind Compass

We now move on to the wind compass; most of the complexity in this template lies in determining whether we need to draw one line or two within the compass.

```
<xsl:template match="@Wind">
<draw:circle draw:style-name="solidLine" draw:layer="layout"
    svg:x="6cm" svg:y="5.5cm"
    svg:width="4cm" svg:height="4cm"/>

<xsl:variable name="dir" select="substring-before(., ',')"/> ❶
<xsl:variable name="dir1">
  <xsl:choose>
  <xsl:when test="contains($dir, '-')"> ❷
    <xsl:value-of select="90 - number(substring-before($dir, '-' ))"/>
  </xsl:when>
  <xsl:otherwise>
    <xsl:value-of select="90 - number($dir)"/>
  </xsl:otherwise>
  </xsl:choose>
</xsl:variable>

<xsl:variable name="dir2"> ❸
  <xsl:choose>
  <xsl:when test="contains($dir, '-')">
     <xsl:value-of select="90 - number(substring-after($dir, '-' ))"/>
  </xsl:when>
  <xsl:otherwise>
     <xsl:value-of select="90 - number($dir)"/>
  </xsl:otherwise>
  </xsl:choose>
</xsl:variable>

<draw:path ❹
    draw:layer="layout"
    svg:viewBox="0 0 2000 100"
    svg:d="m 0 0 2000 0"
    svg:width="2cm" svg:height="0.1cm"
    svg:x="0cm" svg:y="0cm"
    draw:transform="rotate({$dir1 * 3.14159 div 180})
     translate(8cm 7.5cm)"
    draw:style-name="solidLine"/>

<xsl:if test="$dir1 != $dir2">
    <draw:path draw:style-name="dashLine" ❺
        draw:layer="layout"
        svg:viewBox="0 0 2000 100"
        svg:width="2cm" svg:height="0.1cm"
        svg:x="0cm" svg:y="0cm"
        svg:d="m 0 0 2000 0"
        draw:transform="rotate({$dir2 * 3.14159 div 180})
         translate(8cm 7.5cm)"/>
</xsl:if>

<!-- tick marks -->
<draw:path draw:style-name="solidLine" draw:layer="layout"
    svg:x="6cm" svg:y="5.5cm" svg:width="4.5cm" svg:height="4.5cm"
    svg:viewBox="0 0 4500 4500"
    svg:d="M 2000 0 v 125 M 4000 2000 h -125 M 2000 4000 v -125►
 M 0 2000 h 125"/>

<!-- create text near the tick marks --> ❻
<xsl:for-each select="document('')/*/label:compass/label:text">
    <draw:frame draw:style-name="textgraphics" draw:layer="layout"
        svg:width="0.75cm" svg:height=".5cm">
        <xsl:attribute name="svg:x"><xsl:value-of
            select="5.625 + @x"/>cm</xsl:attribute>
        <xsl:attribute name="svg:y"><xsl:value-of
            select="5.25 + @y"/>cm</xsl:attribute>
        <draw:text-box>
            <text:p text:style-name="smallCenter"><xsl:value-of
                select="."/></text:p>
        </draw:text-box>
    </draw:frame>
</xsl:for-each>

<!-- text under the wind compass -->
<draw:frame
    draw:layer="layout"
    svg:x="6cm" svg:y="10cm" svg:height="1cm" svg:width="4cm">
    <draw:text-box>
        <text:p text:style-name="smallCenter">Wind Speed►
(m/sec)</text:p>
        <text:p text:style-name="smallCenter"><xsl:value-of
            select="$dir"/></text:p>
    </draw:text-box>
</draw:frame>
</xsl:template>
```

```
<label:compass> ❼
    <label:text x="2" y="-0.25">N</label:text>
    <label:text x="4.375" y="2">E</label:text>
    <label:text x="2" y="4.25">S</label:text>
    <label:text x="-0.375" y="2">W</label:text>
</label:compass>
```

❶ This gets the direction part of the wind information by grabbing the substring before the separating comma.

❷ If there is a hyphen in the direction, then it must be split into two portions. This sets the first number. The weather report makes north zero degrees and east 90 degrees; OpenDocument says that east is zero and north is 90; hence the formula `90 - n`. If there's no hyphen, then the first direction is simply offset by -90 degrees. The `number` function ensures that string data will be converted to numeric form after stripping leading and trailing whitespace.

❸ Similar code sets the second direction. You may wonder why we didn't use simpler code like this:

```
<xsl:choose>
<xsl:when test="contains($dir,'-')">
    <xsl:variable name="dir1"> ... </xsl:variable>
    <xsl:variable name="dir2"> ... </xsl:variable>
</xsl:when>
<xsl:otherwise>
    <xsl:variable name="dir1"> ... </xsl:variable>
    <xsl:variable name="dir2"> ... </xsl:variable>
</xsl:otherwise>
</xsl:choose>
```

Because a variable exists only within its enclosing block, this code won't work —the `<xsl:when>` or `<xsl:otherwise>` would create the variables, which would disppear immediately upon encountering the ending `</xsl:when>` or `</xsl:otherwise>`. This is why we have to repeat all the choice code within the variable declarations.

❹ This draws the first line; we are guaranteed at least one line. Note the conversion of degrees to radians. To make the transformation easier, we draw the line starting at (0,0), rotate it around the "origin," and then move the rotated line to its final position. The first line is always a solid line.

❺ The second line, if required, is always a dashed line.

❻ We use the same trick of accessing our "lookup table" for the labels and their relative positions.

❼ In these elements, as previously, the offsets are in centimeters.

The Thermometer

Finally, the thermometer. We are drawing the thermometer in two different segments: the straight part and the bulb. We will use a `<draw:circle>` for the bulb because OpenOffice.org doesn't let us use a `<draw:path>` with the `A` (arc) command, and, frankly, the math for a cubic Bézier curve that approximates a circular arc was something that I was totally unwilling to attempt.

```
<xsl:template match="@T">

<xsl:variable name="fillcolor"> ❶
    <xsl:choose>
        <xsl:when test=". &lt; 0">bluefill</xsl:when>
        <xsl:otherwise>redfill</xsl:otherwise>
    </xsl:choose>
</xsl:variable>

<!-- fill the straight part of the thermometer -->
<xsl:variable name="h" select="50*(. + 40)"/> ❷
<draw:polygon draw:style-name="{$fillcolor}" draw:layer="layout"
    svg:x="2.25cm" svg:y="{7.5 - $h div 1000}cm"
    svg:width="0.5cm" svg:height="{$h div 1000}cm"
    svg:viewBox="0 0 500 {$h}"
    draw:points="0,{$h} 0,0 500,0 500,{$h}" />

<!-- fill the bulb -->
<draw:circle draw:style-name="{$fillcolor}" draw:layer="layout"
    draw:kind="cut"
    draw:start-angle="117" draw:end-angle="63"
    svg:x="2cm" svg:y="7.415cm"
    svg:width="1cm" svg:height="1cm"/>

<!-- draw straight part of thermometer --> ❸
<draw:polyline draw:style-name="solidLine" draw:layer="layout"
    svg:x="2.25cm" svg:y="3cm"
    svg:width="0.6cm" svg:height="4.6cm"
    svg:viewBox="0 0 600 4600"
    draw:points="0,4500 0,0 500,0 500,4500" />

<!-- draw the bulb outline -->
<draw:circle draw:style-name="solidLine" draw:layer="layout"
    draw:kind="arc"
    draw:start-angle="117" draw:end-angle="63"
    svg:x="2cm" svg:y="7.415cm"
    svg:width="1cm" svg:height="1cm"/>

<xsl:for-each select="document('')/*/label:thermometer/label:text"> ❹
    <draw:frame draw:style-name="textgraphics" draw:layer="layout"
        svg:width="1cm" svg:height=".5cm">
        <xsl:attribute name="svg:x"><xsl:value-of
            select="1.4 + @x"/>cm</xsl:attribute>
        <xsl:attribute name="svg:y"><xsl:value-of
            select="3 + @y"/>cm</xsl:attribute>
        <draw:text-box>
            <text:p text:style-name="{@tstyle}"><xsl:value-of
                select="."/></text:p>
        </draw:text-box>
    </draw:frame>
```

```
</xsl:for-each>

<draw:frame draw:style-name="textgraphics"
    draw:layer="layout"
    svg:x="1cm" svg:y="8.5cm" svg:height="1cm" svg:width="3cm">
    <draw:text-box>
        <text:p text:style-name="smallCenter">Temp.</text:p>
        <text:p text:style-name="smallCenter"><xsl:value-of
select="."/>
        <xsl:text> / </xsl:text>
        <xsl:value-of select="round((.  div 5) * 9 + 32)"/></text:p>
    </draw:text-box>
</draw:frame>
</xsl:template>

<label:thermometer>
    <label:text x="0" y="-0.175" tstyle="smallRight">50</label:text>
    <label:text x="0" y="2.075" tstyle="smallRight">0</label:text>
    <label:text x="0" y="4.1" tstyle="smallRight">-40</label:text>
    <label:text x="-0.5" y="4.5" tstyle="smallRight">C</label:text>
    <label:text x="1.5" y="-0.175" tstyle="smallLeft">120</label:text>
    <label:text x="1.5" y="2.075" tstyle="smallLeft">32</label:text>
    <label:text x="1.5" y="4.1" tstyle="smallLeft">-40</label:text>
    <label:text x="2" y="4.5" tstyle="smallLeft">F</label:text>
</label:thermometer>
```

❶ Rather than repeat the code for selecting the appropriate color, we store it in a variable.

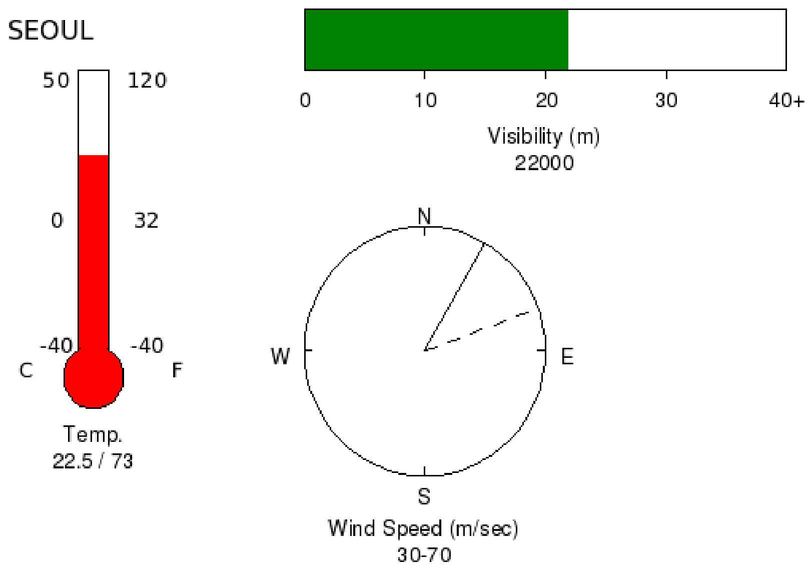
❷ Lots of ugly math here, all to make sure that we make the polygon start at the (0,0) point. This formula will produce ugly results for temperatures above 50 and below –40 celsius.

❸ Again, we draw the outlines after the filled areas to ensure that they are always visible.

❹ Once again, we process `<label:text>` elements to make our work easier. The `x` and `y` attributes are in centimeters, and the `tstyle` attribute gives the text alignment as one of our pre-built styles.

The resulting diagram looks something like Figure 6.9, “Finished Weather Report Drawing”.

Figure 6.9. Finished Weather Report Drawing



If it seems like there is a lot of work to create such a simple drawing, you're right. That's because each object in a drawing is self-contained and needs to carry a lot of information with it.

Before we move on to three-dimensional drawing, we need to cover two more two-dimensional constructs: grouping objects and drawing connectors.

Grouping Objects

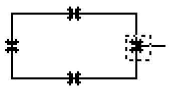
Let's take a breather and look at something far easier—grouping objects. If you want to have objects grouped together (the effect of the *Group* menu item in the *Modify* menu), just put them all inside a `<draw:g>` element. This element is a simple container, without any particular properties of its own. Thus, if you want a transformation to apply to all of the objects in the group, you must apply the transformation to each object individually. Applying the `draw:transform` attribute to the `<draw:g>` element will have no effect.

Grouping is *not* the same as the *Combine* menu item. Combining objects works by creating a single `<draw:path>` element from all of the selected objects.

Connectors

A connector is a line that joins objects and remains attached when you move them. Connectors are attached to "glue points" on the object. The default glue points are at the 12:00, 3:00, 6:00 and 9:00 positions, and are numbered clockwise 0 to 3. Figure 6.10, "Default Glue Points" shows a rectangle with a connector attached to glue point 1.

Figure 6.10. Default Glue Points



Each object joined by a connector has a unique `draw:id`, which is of the XML "ID" type, which must begin with an alphabetic character.

Once the two objects to be joined have their identifying numbers, OpenDocument uses a `<draw:connector>` element with the following attributes that are shared with the `<draw:line>` element:

`draw:style-name`, `draw:text-style-name`
: These attributes refer to `<style:style>` elements; they allow you to have arrows and/or text on the connector.

`draw:layer`
: As with lines, connectors are in the `layout` layer.

`svg:x1`, `svg:y1`, `svg:x2`, `svg:y2`
: The endpoints of the connector.

The following additional attributes complete the connection information:

`draw:type`
: The kind of connector: `standard` (the default, which has 90 degree "corners"), `lines` (multiple straight lines), `line` (a single straight line), or `curve` (curved line).

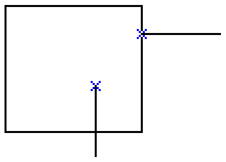
`draw:start-shape`, `draw:end-shape`
: The beginning and ending shape numbers, as determined by their `draw:id` attributes.

`draw:start-glue-point`, `draw:end-glue-point`
: The glue point on the starting and ending objects.

Custom Glue Points

You may add custom glue points to an object. Figure 6.11, “Custom Glue Points” shows a rectangle with two extra glue points, one on the edge and one inside. Example 6.20, “XML for Custom Glue Points” shows the corresponding XML. Each glue point has an integer-valued `draw:id` attribute that gives the glue point number.

Figure 6.11. Custom Glue Points



Example 6.20. XML for Custom Glue Points

```
<draw:rect draw:style-name="gr4" draw:id="id5"
  draw:layer="layout"
  svg:x="3.228cm" svg:y="10.769cm"
  svg:width="1.799cm" svg:height="1.667cm" >
       <draw:glue-point draw:id="5" svg:x="5.002cm" svg:y="-2.777cm"/>
       <draw:glue-point draw:id="6" svg:x="1.623cm" svg:y="1.349cm"/>
</draw:rect>
```

The `svg:x` and `svg:y` attributes are relative values, *not* absolute coordinates. Values range from `-5cm` (the left side or bottom) to `5cm` (the right or top).

You may also set the direction from which a connector exits a glue point by setting the glue point’s `draw:escape-direction` to `left`, `right`, `up`, `down`, `horizontal`, or `vertical`.

Three-dimensional Graphics

Three-dimensional objects are contained in a `<dr3d:scene>` element. It has a large number of attributes that describe the scene’s shading and geometry. Within the scene are one or more `<dr3d:light>` elements, each one describing an illumination source, and one or more three-d shapes, which can be one of the following:

- `<dr3d:cube>`
- `<dr3d:sphere>`
- `<dr3d:rotate>` (an arbitrary path rotated about a central axis)
- `<dr3d:extrude>` (an arbitrary path extruded into a third dimension)

The dr3d:scene element

This element has the standard position-and-bounds attributes: `svg:x`, `svg:y`, `svg:width`, and `svg:height`. Along with these come the attributes specifically designed for three-dimensional objects. Many of these will specify *vectors*, which come in the form of an *x*-, *y*-, and *z*-coordinate in parentheses, separated by whitespace. OpenDocument's coordinate system is a left-handed system: the positive *z* axis points out of the screen towards the person viewing the drawing. The values are measured in units of 1000 per centimeter.

`dr3d:vrp`
: The origin of the viewing volume. This is not the same as the camera position. That is set by

`dr3d:distance`, `dr3d:focal-length`
: These are both measured in centimeters.

`dr3d:vpn`
: Direction the camera is pointing

`dr3d:vup`
: "Which way is up?" This attribute tells you. The default is the vector `(0 6760000 0)`.

`dr3d:projection`
: This attribute has the value `perspective` (the default) or `parallel`. In perspective projection, objects get smaller as they become more distant.

`dr3d:shadow-slant`
: A non-negative integer; default value is zero. No matter what the setting of this attribute, you will not see a shadow unless the object's `<style:style>` element has `dr3d:shadow` set to `visible`.

`dr3d:shade-mode`
: The default value is `gouraud`, which is the most "lifelike" shading. `phong` shading somewhat less so, and `flat` shading lets you see the polygons that make up the object in all their individual majesty.

`dr3d:ambient-color`
: The ambient light color, as a hex value.

`dr3d:lighting-mode`
: This boolean attribute has a default value of `false`. This attribute does not appear to have any effect.

Lighting

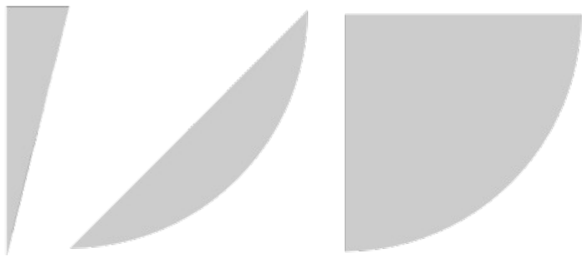
The `<dr3d:scene>`'s first eight children are `<dr3d:light>` elements. Each element describes a light source that illuminates the object if its `dr3d:enabled` attribute is `true`. A light source has a `dr3d:diffuse-color`, the color of the light as a hex value; a `dr3d:direction`, a vector with default value is directly in front of the object— `(0 0 1)`; and `dr3d:specular`, a boolean which tells whether you want reflective highlights from this light source. The first light source will have this value set to `true`; other light sources will have a value of `<false>`.

The Object

Following the lighting is the description of the three-d object itself. Cubes and spheres have their own special elements: `<dr3d:cube>` and `<dr3d:sphere>`. These need no attributes other than the `draw:layer` and controling `draw:style-name`. All other objects are described by a generic `<dr3d:rotate>` element, which describes a path exactly as described in the section called "Polylines, Polygons, and Free Form Curves", with a `svg:viewBox` and `svg:d` attribute.

The path for a cylinder is a rectangle. The path for a cone is a triangle; note that the triangle is "upside-down." The path for a torus is a circle, the path for a shell is a chord, and the path for a half-sphere is a quarter-circle, as shown in Figure 6.12, "Path for a Cone, Shell, and Half-Sphere". When you convert an arbitrary path to a rotation object, your path becomes the path of the `<dr3d:rotate>` object.

Figure 6.12. Path for a Cone, Shell, and Half-Sphere



Example 6.21, "XML for a Cylinder" shows the XML for a cylinder; only the single enabled light source is included.

Example 6.21. XML for a Cylinder

```
<dr3d:scene draw:name="cylinder" draw:style-name="gr5"
  svg:width="0.795cm" svg:height="1.589cm"
  svg:x="7.646cm" svg:y="9.26cm"
  dr3d:vrp="(0 0 16885.7142857143)" dr3d:vpn="(0 0 14285.7142857143)"
  dr3d:vup="(0 6760000 0)"
  dr3d:projection="perspective"
  dr3d:distance="2.6cm" dr3d:focal-length="10cm"
  dr3d:shadow-slant="0" dr3d:shade-mode="gouraud"
  dr3d:ambient-color="#666666" dr3d:lighting-mode="false">
```

```
    <dr3d:light
      dr3d:diffuse-color="#cccccc"
      dr3d:direction="(0.57735026918963 0.57735026918963
        0.57735026918963)"
      dr3d:enabled="true" dr3d:specular="true"/>
    <dr3d:rotate draw:style-name="gr6" draw:layer="layout"
      svg:viewBox="0 -5000 2500 10000"
      svg:d="m0-5000h250 250 500 500 500 250
        250v10000h-250-250-500-500-500-250-250"/>
</dr3d:scene>
```

Extruded Objects

The final method of creating a three-dimensional object is by *extrusion* (as if the path were squeezed out of a toothpaste tube). An extruded object is represented by a `<dr3d:extrude>` object within the scene. It also contains `svg:viewBox` and `svg:d` attributes that define the coordinate system and path to be extruded.

Note

When you convert text to extruded three-dimensional objects, each letter is converted to a path which becomes a separate extruded object. The letters lose their identity as text, and are no longer editable as text.

Styles for 3-D Objects

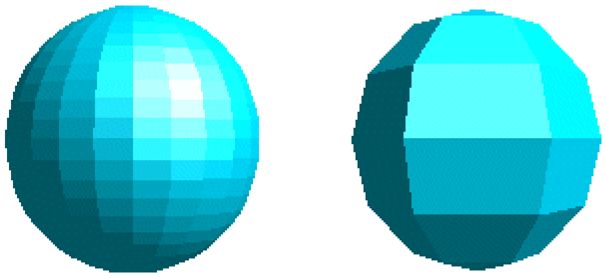
As with all other drawing objects, the presentation is affected by a `<style:style>` referred to by the object's `draw:style-name` property. Both the scene and the object have a `draw:style-name`, but only the one attached to the object controls the presentation—the one attached to the scene is ignored.

As with two-dimensional objects, you specify the color with `draw:fill-color` for solid colors, `draw:gradient` for a color range, `draw:hatch` for patterns, or `draw:fill-image` for bitmaps.

In addition to the two-dimensional drawing information, you may apply the following three-dimensional attributes:

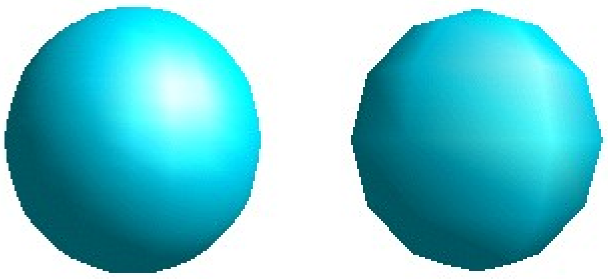
`dr3d:horizontal-segments`, `dr3d:vertical-segments`
: The number of horizontal segments tells how many "slices" to stack up to create an object. The number of vertical segments tells how many degrees to take when it rotates an object. Figure 6.13, "Flat-shaded Spheres with Differing Numbers of Segments" shows a sphere drawn in two ways; the one on the left uses the default values of 24 horizontal segments and 24 vertical segments (i.e., rotation at 15 degree intervals).

Figure 6.13. Flat-shaded Spheres with Differing Numbers of Segments



The sphere on the right has only six horizontal segments and six vertical segments (rotation at 60 degree intervals). The figure uses flat shading to clearly display the difference. When using Gouraud shading, as shown in Figure 6.14, "Gouraud-shaded Spheres with Differing Numbers of Segments", the default values assume a smooth spherical appearance.

Figure 6.14. Gouraud-shaded Spheres with Differing Numbers of Segments



Note

OpenOffice.org actually uses the same path information to generate a cone and a pyramid; the cone has 24 vertical segments and the pyramid has four, giving it a rectangular base.

`dr3d:end-angle`

This number, in tenths of a degree, tells where to stop the rotation about the *y*-axis. The default value is 3600. Rotation proceeds counterclockwise "into" the screen. Thus, a half-cone will show the sheared-off face with the rounded area behind it.

`dr3d:back-scale`

This attribute should be applied only to extruded objects. It gives the ratio of the size of the front of the object to the size of the back of the object. Thus, if the value is `50%`, the front face of the object will be half the size of the back face.

`dr3d:depth`
: This attribute's value, given in centimeters (such as `1.25cm`, tells the depth of an extruded object. It has no effect on rotated objects.

`dr3d:normals-kind`, `dr3d:normals-direction`
: In these attributes, the word *normal* doesn't mean "default"; it means "perpendicular to a surface." The `dr3d:normals-kind` attribute tells how light reflects off an object. A value of `flat` shows individual polygons, `sphere` renders a smooth surface, and `object` chooses the best method depending upon the object. You may invert the light source (absorbing instead of reflecting) by setting `<dr3d:normals-direction>` to `inverse` The other value is `normal`, which in this case *does* mean "the ordinary way."

`dr3d:diffuse-color`, `dr3d:emissive-color`, `dr3d:specular-color`
: All of these attributes take a hexadecimal color specification as their value, and describe the behavior of the "material" that the the object is made of. `dr3d:diffuse-color` is the color of light from outside that scatters when reflected. `dr3d:emissive-color` is the color of light that the object itself emits. `dr3d:specular-color` is the color of reflective highlights.

`dr3d:shininess`
: This is a percent value which appears to be implemented in the opposite way that its name implies. A shininess of `100%` produces a non-reflective object; a shininess of `0%` produces a highly reflective one.

`dr3d:close-front`, `dr3d:close-back`
: Apply these boolean attributes to extruded objects. Specify `true` if you want the front and/or back face of the object to be displayed, `false` if you don't.

`dr3d:shadow`
: The default value for this attribute is `hidden`; if set to `visible`, a three-d shadow appears. This attribute overrides the `draw:shadow` attribute.

`style:repeat`
: If you are using a bitmap for your object, you may set this attribute to `repeat` to tile the surface or `stretch` to wrap the object in the bitmap.

`dr3d:texture-kind`
: Another attrbute for textured objects (bitmap fill) that seems to be contradictory: You may select `color`, which produces a black-and-white effect. Selecting `luminance`, the default, shows the bitmap in color.

`dr3d:texture-mode`
: When set to `blend`, you see both the bitmap and the fill color (where the bitmap is transparent). When set to `modulate`, you see only the bitmap.

`dr3d:texture-generation-mode-x`, `dr3d:texture-generation-mode-y`
: These attributes determine how the texture coordinates are generated when wrapping a bitmap around an object. The possible values are `object` (object-dependent), `parallel`, and `sphere`.

There are two other three-d attributes whose settings appear to have no effect on the picture: `dr3d:backface-culling` and `dr3d:texture-filter`. Both these attributes may either be `enabled` or `disabled`.

Chapter 7. Presentations

Much of what you need to know about presentations has already been covered in the preceding chapter, because a presentation is at heart a series of `<draw:page>` elements within the `<office:presentation>` element.

> **Note**
> In fact, you can create multiple pages in a drawing document. They are simply stored as separate pages, and don't have any of the transition elements that a presentation permits.

Presentation Styles in `styles.xml`

In OpenOffice.org, there's so much information (53K bytes worth) in the `styles.xml` file for even the simplest of presentations that you probably don't want to create it from scratch. Instead, you will be better off to create an empty presentation with the background elements that you want, and then merge your own `content.xml` into the resulting jar file. Here are some of the names of `<style:style>`s that are in the `styles.xml` file for a plain presentation, along with their salient characteristics. These are styles for OpenOffice.org; for other applications "your mileage may vary."

Table 7.1. Default Presentation Styles

Style name	Characteristics
`standard`	a 18-point sans-serif font; specifies bulleted lists with 0.6cm indenting at each level. All the other styles are based on this one.
`textbody`	a 16-point font.
`title`	a 44-point font.
`title1`	a 24-point shadowed font.
`title2`	a 36-point shadowed font with a bulleted list.
`headline`, `headline1`, `headline2`	a 24-point, 18-point, and 14-point font.
`measure`	a 12-point font with arrow markers at both ends for lines.
`Default-title`	This is the first of the styles with `style:family` equal to `presentation`; the others are in the `graphic` family. Specifies a 44-point serif font with bullets at 0.6cm indenting for each level.
`Default-subtitle`	A 32-point serif font with bullets at 0.6cm indenting for each level.
`Default-background`	If your master slide has a background fill color, gradient, or image, this is where you specify it.

Style name	Characteristics
`Default-backgroundobjects`	This style is the parent style for any objects that are on the master slide.
`Default-outline1..9`	These nine styles are used for display in outline mode. `Default-outline1` has 32-point serif font with bullets at unusual intervals: 0.3cm, 1.6cm, 3cm, and then at 1.2cm spacing. `Default-outline2` is 28-point with a text indent of -0.8 centimeters. `Default-outline3` is 24-point with a text indent of -0.6 centimeters, and the remaining styles are 20-point. Each one inherits from its predecessor; thus, `Default-outline5`'s `style:parent-style-name` is `Default-outline-4`.

Page Layouts in `styles.xml`

These `<style:style>` elements are followed by `<style:presentation-page-layout>` elements, one for each different kind of page layout that you use. These are *incredibly* important; if you want your document displayed correctly, you must have these page layout styles in the `styles.xml` file.

Each `<style:presentation-page-layout>` has a `style:name` attribute. There is one `<presentation:placeholder>` for each visual element on the slide master. This placeholder element has a `presentation:object` attribute that tells what kind of object you are dealing with: `title`, `subtitle`, `text`, `outline`, `graphic`, `table` (spreadsheet), `object`, `chart`, and `orgchart`. The value of `handout` is used to define the handout format; the values `page` and `notes` are not used in this context. Each `<presentation:placeholder>` has its location and size specified with `svg:x`, `svg:y`, `svg:width`, and `svg:height` attributes.

Master Styles in `styles.xml`

The `<office:automatic-styles>` is followed by `<office:master-styles>`. Just as described in the section called “A Drawing’s styles.xml File”, the `<office:master-styles>` begins with a `<draw:layer-set>` element.

In a presentation, the layer set is followed by a `<style:handout-master>` element. Its `presentation:presentation-page-layout-name` attribute refers to the `style:presentation-page-layout` whose `<presentation:placeholder>`s describe handouts. The `<style:handout-master>` starts with some `<draw:frame>` elements that give the `<presentation:header>`, `<presentation:date-time>`, and `<presentation:footer>`. These text frames are followed by one `<draw:page-thumbnail>` element for each thumbnail; the values of the thumbnail’s `svg:x`, `svg:y`, `svg:width`, and `svg:height` attributes are different from those of the `<presentation:placeholder>`, and are the ones that are actually used when displaying the thumbnail.

The next master style is a `<style:master-page>` element which describes the layout for the default slide type (text with outline) notes view. It contains a `<draw:frame>` elements for a default title, an outline area, date and time, footer, and page number (which is enclosed in a `text:page-number` element inside the frame's text box). These frames are followed by a `<presentation:notes>` element, which contains a `<draw:page-thumbnail>` element for the page and then various `<draw:frames>` to describe the rest of the page.

Example 7.1, "Structure of a styles.xml file" shows a `styles.xml` file, with much non-essential information removed. Note how the names of the `<style:page-master>` elements are referenced in the `<style:handout-master>` and `<style:master-page>`; they are shown in boldface. Note also that the similar names of the `<style:page-master>` and `<style:master-page>` elements doesn't make this any easier.

Example 7.1. Structure of a styles.xml file

```
<office:styles>
    <draw:marker draw:name="Arrow" svg:viewBox="0 0 20 30"
      svg:d="m10 0-10 30h20z"/>
    <style:default-style style:family="graphic">
        <style:paragraph-properties several properties>
            <style:tab-stops/>
        </style:paragraph-properties>
        <style:text-properties font declarations/>
    </style:default-style>
    <style:style style:name="standard"
      style:family="graphic">
        <style:graphic-properties many properties>
            <text:list-style>
                <text:list-level-style-bullet text:level="1"
                  text:bullet-char="●">
                    <style:text-properties fo:font-family="StarSymbol"
                      style:use-window-font-color="true"
                      fo:font-size="45%"/>
                </text:list-level-style-bullet>
                <!-- and nine more bullets -->
            </text:list-style>
        </style:graphic-properties>
        <style:paragraph-properties many specifications/>
    </style:style>
    <!-- we have omitted many other default styles -->

    <style:presentation-page-layout style:name="AL0T26">
        <presentation:placeholder
          presentation:object="handout"
          svg:x="2.057cm" svg:y="1.743cm"
          svg:width="6.103cm" svg:height="-0.233cm"/>
        <presentation:placeholder
          presentation:object="handout"
          svg:x="10.96cm" svg:y="1.743cm"
          svg:width="6.103cm" svg:height="-0.233cm"/>
```

```
            <!-- etc -->
        </style:presentation-page-layout>
</office:styles>

<office:automatic-styles>
    <!-- page master for handouts -->
    <style:page-layout style:name="PM0">
        <style:page-layout-properties
            fo:margin-top="0cm" fo:margin-bottom="0cm"
            fo:margin-left="0cm" fo:margin-right="0cm"
            fo:page-width="27.94cm" fo:page-height="21.59cm"
            style:print-orientation="landscape"/>
    </style:page-layout>

    <!-- page master for ordinary pages -->
    <style:page-layout style:name="PM1">
        <style:page-layout-properties
            fo:margin-top="0cm" fo:margin-bottom="0cm"
            fo:margin-left="0cm" fo:margin-right="0cm"
            fo:page-width="28cm" fo:page-height="21cm"
            style:print-orientation="landscape"/>
    </style:page-layout>

    <!-- page master for notes -->
    <style:page-layout style:name="PM2">
        <style:page-layout-properties
            fo:margin-top="0cm" fo:margin-bottom="0cm"
            fo:margin-left="0cm" fo:margin-right="0cm"
            fo:page-width="21.59cm" fo:page-height="27.94cm"
            style:print-orientation="portrait"/>
    </style:page-layout>

    <!-- style for the default page -->
    <style:style style:name="dp1" style:family="drawing-page">
        <style:drawing-page-properties
            draw:background-size="border"
            draw:fill="none"/>
    </style:style>
</office:automatic-styles>

<office:master-styles>
    <draw:layer-set>
        <draw:layer draw:name="layout"/>
        <draw:layer draw:name="background"/>
        <draw:layer draw:name="backgroundobjects"/>
        <draw:layer draw:name="controls"/>
        <draw:layer draw:name="measurelines"/>
    </draw:layer-set>

    <style:handout-master
      presentation:presentation-page-layout-name="AL0T26"
      style:page-layout-name="PM0" draw:style-name="dp2">
        <draw:frame draw:style-name="gr1" draw:text-style-name="P1"
          draw:layer="backgroundobjects"
          svg:width="12.124cm" svg:height="1.078cm"
          svg:x="0cm" svg:y="0cm" presentation:class="header">
            <draw:text-box>
                <text:p text:style-name="P1">
                    <presentation:header/>
                </text:p>
            </draw:text-box>
        </draw:frame>
        <!-- frames for date/time and footer follow -->
        <draw:page-thumbnail
            draw:layer="backgroundobjects"
            svg:width="5.586cm" svg:height="4.189cm"
            svg:x="2.794cm" svg:y="4.327cm" draw:page-number="1"/>
        <draw:page-thumbnail draw:layer="backgroundobjects"
            svg:width="5.586cm" svg:height="4.189cm"
            svg:x="11.176cm" svg:y="4.327cm"/>
    </style:handout-master>

    <style:master-page style:name="Default"
      style:page-layout-name="PM1"
      draw:style-name="dp1">
        <draw:frame presentation:style-name="Default-title"
          draw:layer="backgroundobjects"
          svg:width="25.199cm" svg:height="3.506cm"
          svg:x="1.4cm" svg:y="0.837cm"
          presentation:class="title" presentation:placeholder="true">
            <draw:text-box/>
        </draw:frame>

        <presentation:notes style:page-layout-name="PM2">
            <draw:page-thumbnail
              presentation:style-name="Default-title"
              draw:layer="backgroundobjects"
              svg:width="13.968cm" svg:height="10.476cm"
              svg:x="3.81cm" svg:y="2.123cm"
              presentation:class="page"/>
            <draw:frame presentation:style-name="Default-notes"
              draw:layer="backgroundobjects"
              svg:width="17.271cm" svg:height="12.572cm"
              svg:x="2.159cm" svg:y="13.271cm"
              presentation:class="notes"
              presentation:placeholder="true">
                <draw:text-box/>
            </draw:frame>
            <!-- frame for header and date/time, footer, and page
                 number follow -->
        </presentation:notes>
    </style:master-page>
</office:master-styles>
```

A Presentation’s `content.xml` File

In the `content.xml` file’s `<office:presentation>` element you will find a series of `<draw:page>` elements, each of which has a `draw:name` attribute in the form `page`*n*. The `draw:master-page-name` and `draw:style-name` attributes refer to the master page and style for the slide. These will normally be the same for all slides. The `presentation:presentation-page-layout-name` attribute points to the default layout from the `styles.xml` file.

The `<draw:page>` element contains child elements for each of the objects on the slide. These can be

- `<draw:frame>` for a title, subtitle, outline, or a "placeholder."
- `<draw:image>` for clip art.
- `<draw:object>` for embedded charts or spreadsheets.

These elements have the following attributes in common: `draw:layer` is set to `layout`. If there is a `presentation:class` attribute, it tells what kind of item this is: `title`, `outline`, `subtitle`, or `notes`. The item's location and size is given by `svg:x`, `svg:y`, `svg:width`, and `svg:height` attributes. The element will also have a `draw:style-name` attribute that refers to a style in the `<office:automatic-styles>` section of the file.

Warning

The layout of a page is controlled by the `style.xml`'s `<style:presentation-page-layout>` elements as described in the section called "Page Layouts in styles.xml". If you set the location and size attributes for a particular element in the `content.xml` file, they will not take effect unless you also set the `presentation:user-transformed` to `true`.

If a slide doesn't have content yet (it shows up as "Click for title/text/chart" in OpenOffice.org), that means that the element has its `presentation:placeholder` attribute set to `true`.

Text Boxes in a Presentation

The `<draw:frame>` for text boxes has additional `presentation:style-name` and `draw:text-style-name` attributes. The children of the `<draw:text-box>` element will normally be `<text:p>` elements for titles and `<text:list>` elements for outlines. (Of course, if you add anything other than the defaults, such as an ordered list, that will also go into the `<draw:text-box>`. Example 7.2, "Text Boxes for a Page with Title and Two Outlines" shows the text boxes for a page with a title and a left and right outline, each containing one item.

Example 7.2. Text Boxes for a Page with Title and Two Outlines

```
<draw:frame presentation:style-name="pr1"
  draw:text-style-name="P1" draw:layer="layout"
  svg:width="25.191cm" svg:height="3.508cm"
  svg:x="1.399cm" svg:y="0.837cm" presentation:class="title">
    <draw:text-box>
        <text:p text:style-name="P1">Title</text:p>
    </draw:text-box>
</draw:frame>

<draw:frame presentation:style-name="pr4"
  draw:text-style-name="P4" draw:layer="layout"
  svg:width="12.292cm" svg:height="13.864cm"
  svg:x="1.399cm" svg:y="4.915cm" presentation:class="outline">
```

```
    <draw:text-box>
        <text:list text:style-name="L2">
            <text:list-item>
                <text:p text:style-name="P4">Left outline</text:p>
            </text:list-item>
        </text:list>
    </draw:text-box>
</draw:frame>

<draw:frame presentation:style-name="pr7" draw:layer="layout"
  svg:width="12.292cm" svg:height="13.613cm"
  svg:x="14.306cm" svg:y="4.915cm" presentation:class="outline">
    <draw:text-box>
        <text:list text:style-name="L2">
            <text:list-item>
                <text:p text:style-name="P4">Right outline</text:p>
            </text:list-item>
        </text:list>
    </draw:text-box>
</draw:frame>
```

Images and Objects in a Presentation

Clip art is represented by a `<draw:image>` element, with attributes exactly as described in the section called “Body Information for Images in Text”. Embedded spreadsheets and charts are represented by a `<draw:object>` element. Images and objects have these attributes in common:

- `xlink:href`, whose value is a local URL
- `xlink:type`, always set to `simple`
- `xlink:show`, always set to `embed`, and
- `xlink:actuate`, always set to `onLoad`

Example 7.3, “Clip Art and Spreadsheet in a Presentation” shows the XML for clip art and an embedded spreadsheet.

Example 7.3. Clip Art and Spreadsheet in a Presentation

```
<draw:frame draw:style-name="gr2" draw:text-style-name="P7"
  draw:layer="layout"
  svg:width="11.205cm" svg:height="7.327cm"
  svg:x="12.925cm" svg:y="4.738cm">
    <draw:image
      xlink:href="Pictures/100000000000021500000190BC13A5A7.jpg"
      xlink:type="simple" xlink:show="embed" xlink:actuate="onLoad">
        <text:p text:style-name="P1"/>
    </draw:image>
</draw:frame>

<draw:frame draw:name="Object 2" draw:style-name="standard"
  draw:layer="layout"
  svg:width="25.191cm" svg:height="13.856cm"
  svg:x="1.399cm" svg:y="4.915cm"
  presentation:class="table" presentation:user-transformed="true">
    <draw:object xlink:href="./Object 3" xlink:type="simple"
      xlink:show="embed" xlink:actuate="onLoad"/>
</draw:frame>
```

```
<draw:frame draw:name="Object 3" draw:style-name="standard"
  draw:layer="layout"
  svg:width="21.797cm" svg:height="10.079cm"
  svg:x="5.081cm" svg:y="6.174cm"
  presentation:class="chart" presentation:user-transformed="true">
    <draw:object xlink:href="./Object 2" xlink:type="simple"
      xlink:show="embed" xlink:actuate="onLoad"/>
</draw:frame>
```

Finally, each slide can have notes which are represented by a `<presentation:notes>` element with two children. The `<draw:page-thumbnail>` child element gives the size and position of the slide thumbnail, and has a `presentation:class` of `page`. The `draw:page-number` attribute tells which slide should be thumbnailed. The following `<draw:frame>` element's child `<draw:text-box>` contains (naturally enough) the notes for the slide, and has a `presentation:class` of `notes`. Example 7.4, "Presentation Notes" shows the relevant XML.

Example 7.4. Presentation Notes

```
<presentation:notes draw:style-name="dp2">
    <draw:page-thumbnail draw:style-name="gr1" draw:layer="layout"
      svg:width="13.968cm" svg:height="10.476cm"
      svg:x="3.81cm" svg:y="2.123cm" draw:page-number="5"
      presentation:class="page"/>
    <draw:frame presentation:style-name="pr5"
      draw:text-style-name="P3" draw:layer="layout"
      svg:width="17.271cm" svg:height="12.573cm"
      svg:x="2.159cm" svg:y="13.271cm"
      presentation:class="notes">
        <draw:text-box>
            <text:p text:style-name="P3">Take note of this▶
slide.</text:p>
        </draw:text-box>
    </draw:frame>
</presentation:notes>
```

Text Animation

To insert animated text into a slide, set the following attributes in the `<draw:frame>`'s controlling style:

`text:animation`
: The type of animation. The valid values are given here, along with the corresponding choice in the OpenOffice.org application dialog box: `none`, `scroll` (scroll through), `alternate` (scroll back and forth), and `slide` (scroll in).

`text:animation-direction`
: `left`, `right`, `up`, or `down`

`text:animation-start-inside`, `text:animation-stop-inside`
: Do you want the text to be visible at the beginning or end of the animation? Set these attributes to `true` or `false` accordingly.

`text:animation-repeat`
: If you want the animation repeated a specific number of times, set this attribute's value to a positive integer. If you want continuous animation, set the value to zero.

`text:animation-steps`
: The distance that the text moves at each iteration, expressed as a *length*, such as `0.254cm` or `5px`.

`text:animation-delay`
: The amount of time between animation steps, expressed as a duration. Two and a half seconds would be written as `PT00H00M02,5S`. Note the use of a comma instead of a decimal point. For "automatic," use `PT00H00M00S`

`style:text-blinking`
: Finally—and the subject is not a pleasant one—it is possible to create blinking text by setting this attribute's value to `true`. In addition to irritating the people who view your presentation, this attribute will override any setting of `text:animation`.

SMIL Animations

It is also possible to use elements borrowed from the Synchronized Multimedia Integration Language (SMIL) specification to animate text on a slide. These elements come after the text and before the notes. A full discussion of SMIL is well beyond the scope of this book, so we will just show Example 7.5, "Simple SMIL animation", which is the XML for an animation that has text fly in from the left:

Example 7.5. Simple SMIL animation

```
<anim:par presentation:node-type="timing-root"> ❶
    <anim:seq presentation:node-type="main-sequence">
        <anim:par smil:begin="next">
            <anim:par smil:begin="0s">
                <anim:par smil:begin="0s"
                  smil:fill="hold" ❷
                  presentation:preset-property=
                    "Direction;Accelerate;Decelerate" ❸
                  presentation:node-type="on-click"
                  presentation:preset-class="entrance"
                  presentation:preset-id="ooo-entrance-fly-in"
                  presentation:preset-sub-type="from-left">
                    <anim:set smil:begin="0s" ❹
                      smil:dur="0.0015s" smil:fill="hold"
                      smil:targetElement="id1"
                      smil:attributeName="visibility"
```

```
                        smil:to="visible"/>
                      <anim:animate smil:dur="0.75s"
                        smil:fill="hold"  ❺
                        smil:targetElement="id1" smil:attributeName="x"
                        smil:values="0-width/2;x" smil:keyTimes="0;1"
                        presentation:additive="base"/>
                      <anim:animate smil:dur="0.75s" smil:fill="hold"
                        smil:targetElement="id1" smil:attributeName="y"
                        smil:values="y;y" smil:keyTimes="0;1"
                        presentation:additive="base"/>
                  </anim:par>
              </anim:par>
          </anim:par>
      </anim:seq>
</anim:par>
```

❶ The children of the `anim:par` element are all animated in parallel; the children of the `anim:seq` element are animated in sequence.

❷ The `smil:fill` has nothing to do with filling areas with colors; it tells the animation engine how to fill the time after part of the animation is done. The value of `hold` means to hold the animated object at its ending position rather than remove it.

❸ If you are building a presentation programmatically, you do not need to include these attributes. The presentation will display properly, but the application will not be aware that any animation has been associated with the slide when you edit it.

❹ The first part of the animation simply makes the object appear.

❺ The `anim:animate` elements move the *x* coordinate in increasing values while leaving *y* unchanged. The `smil:keyTimes` give the “breakpoints” for the animation as floating point numbers, where 0 is the beginning of the duration and 1 is the end.

Transitions

Transitions from one slide to the next are controlled by attributes in the `style:style` that is associated with a slide (it will have a `style:family="drawing-page"` attribute).

Warning

If you are creating a slideshow of your own for OpenOffice.org, the `draw:style-name` *must* have the form `dp`*n*, where *n* is an integer. Using just any old name as an identifier will not work.

The `<style:style>` element will contain a `<style:drawing-page-properties>` element that sets the `presentation:transition-speed`, `smil:type` and `smil:subtype` attributes. The speed may be one of `slow`, `medium`, or `fast`. There are *lots* of transition types and subtypes, fully summarized in the SMIL recommendation at `http://www.w3.org/TR/SMIL/`. Here is a table of just a few of the possibilities:

Table 7.2. Some of the Many SMIL Transition Types

smil:type	**smil:subtype**
`barWipe`	`leftToRight`, `topToBottom`
`boxWipe`	`topLeft`, `topRight`, `bottomRight`, `bottomLeft`, `topCenter`, `rightCenter`, `bottomCenter`, `leftCenter`
`irisWipe`	`rectangle` `diamond`
`fade`	`crossfade`, `fadeToColor`, `fadeFromColor`. The first of these will also have a `smil:fadeColor` attribute.
`waterfallWipe`	`verticalLeft`, `verticalRight`, `horizontalLeft`, `horizontalRight`
`pinWheelWipe`	`twoBladeVertical`, `twoBladeHorizontal`, `fourBlade`

To include a sound with a slide transition, you insert a `<presentation:sound>` element in the slide's controlling `<style:style>`. The `<presentation:sound>` has an `xlink:href` whose value is the path to the sound file. Other attributes should be set as `xlink:type="simple"`, `xlink:show="new"`, and `xlink:actuate="onRequest"`.

Interaction in Presentations

If you wish to have a `draw:frame` respond to a mouse click, add code like this between the opening and closing tags of the object:

```
<office:event-listeners>
    <presentation:event-listener script:event-name="dom:click"
     presentation:action="action-to-perform"/>
</office:event-listeners>
```

The `presentation:action` attribute can have a value of `none` (the default), `previous-page`, `next-page`, `first-page`, `last-page`, `hide` (hides the object), `fade-out` (fade out this object), or `stop` (exits the slideshow). These values require no additional information.

The presentation:action values of show (display a specific slide), execute (run a program), and sound require extra information. Running a macro in response to a mouse click is done in an entirely different manner altogether, and, since we have not covered scripting in this book, we will not discuss that option. Example 7.6, “XML for Presentation Actions Requiring Extra Information” shows the XML for showing slides, running programs, and playing sounds in response to a mouse click. In this example, all the actions are connected with text boxes. The position and size attributes for the frames are not shown. In the code for playing a sound, note that the <presentation:sound> element becomes a child of the <presentation:event-listener> element.

Example 7.6. XML for Presentation Actions Requiring Extra Information

```
<!-- Show a specific slide -->
<draw:frame draw:style-name="gr2" draw:layer="layout">
    <draw:text-box>
        <text:p text:style-name="P1">GO TO SLIDE 3</text:p>
    </draw:text-box>
    <office:event-listeners>
        <presentation:event-listener script:event-name="dom:click"
          presentation:action="show"
          xlink:href="#page3"
          xlink:type="simple" xlink:show="new"
          xlink:actuate="onRequest"/>
    </office:event-listeners>
</draw:frame>

<!-- Run a program -->
<draw:frame draw:style-name="gr2" draw:layer="layout">
    <draw:text-box>
        <text:p text:style-name="P1">RUN PROGRAM</text:p>
    </draw:text-box>
    <office:event-listeners>
        <presentation:event-listener script:event-name="dom:click"
          presentation:action="execute"
          xlink:href="/opt/kde3/bin/kcalc"
          xlink:type="simple" xlink:show="new
          xlink:actuate="onRequest"/>
    </office:event-listeners>
</draw:frame>

<!-- Play a sound -->
<draw:frame draw:style-name="gr2" draw:layer="layout">
    <draw:text-box>
        <text:p text:style-name="P1">SOUND</text:p>
    </draw:text-box>
    <office:event-listeners>
        <presentation:event-listener script:event-name="dom:click"
          presentation:action="sound">
            <presentation:sound
              xlink:href="/usr/local/openoffice/share/gallery/►
sounds/applause.wav"
              xlink:type="simple" xlink:show="new"
              xlink:actuate="onRequest"/>
        </presentation:event-listener>
    </office:event-listeners>
</draw:frame>
```

Case Study: Creating a Slide Show

In this case study, we will use Perl to create (literally) a slide show from a series of photographs such as one might take on vacation.[12] The input file is a simple text file consisting of alternating file names and descriptions:

```
dscn0134_a.jpg
Rubies at Smithsonian Institution
dscn0157_a.jpg
Mt. Rushmore
dscn0181_a.jpg
Devil's Tower National Monument
dscn0210_a.jpg
Buffalo at Yellowstone
dscn0242_a.jpg
Old Faithful
dscn0274_a.jpg
Grand Teton
dscn0329_a.jpg
Baby Llama
```

Our slide show will place the description at the top of the page and the picture beneath it. We will have “Next” and “Back” buttons at the bottom of each slide, with the exception of the first and last slides, which won’t let you fall off the edge of the earth. We will also have transitions for the slides; they will move onto the screen from the left.

The program starts off with variable declarations, and grabs two arguments from the command line: the file name containing the picture list, and an output file name. [You will find the pictures, the picture list (file `piclist.txt`), and the program (file `make_slideshow.pl`) in directory `ch07` in the downloadable example files.]

```
#!/usr/bin/perl
use strict;
use warnings;

use Archive::Zip;
use File::Spec;

my  $file_name;         # file name to insert in slide show
my  $description;       # description of the slide
my  ($width, $height);  # height and width of a picture
my  $left_edge;         # where left edge of picture begins

my  $archive;           # zip archive
my  $content_filename;  # name of content file
my  $styles_filename;   # name of styles file
my  @piclist;           # stores picture file names and descriptions

my  $i;                 # ubiquitous loop counter
my  $slide_number;      # current slide number
```

[12] Thanks to my brother Steve, who provided photos from his vacation for the sample files.

```
#
#   This program requires two arguments: the picture list
#   and the output file name.

if (scalar @ARGV != 2)
{
    print "Usage: $0 picturelist outputFilename\n";
    exit;
}
```

Because we don’t know how many pictures there are *a priori*, and we need to do something special on the last picture, we will have to read all the picture names and descriptions into an array for future use:

```
#   Read the picture file names and their descriptions
#   into an array.
#
open INFILE, "<$ARGV[0]" or die("Cannot find file $ARGV[0]");
while (<INFILE>)
{
    chomp;
    next if (m/^\s*$/); # skip blank lines
    push @piclist, $_;
}
close INFILE;
```

The code continues by setting up a new .ZIP archive and adding the subdirectory for the images. We call `File::Spec->tmpdir()` to provide a temporary directory for storing the `META-INF/manifest.xml`, `styles.xml`, and `content.xml` that we will be creating. To make the program easier to read, we separate the creation of the manifest and style files into separate subroutines. The styles file has constant contents, so we will leave it for last.

```
#
#   Create a .ZIP archive
#
$archive = Archive::Zip->new( );

#
#   Add a directory for the images
#
$archive->addDirectory( "Pictures" );

#
#   create a temporary file for the manifest,
#   create the META-INF/manifest.xml file, and
#   add it to the archive
#
$manifest_filename = File::Spec->catfile(File::Spec->tmpdir(),
    "manifest.xml");
open MANIFESTFILE, ">$manifest_filename" or
    die("Cannot open $manifest_filename");

create_manifest_file( );

close MANIFESTFILE;
```

```
$archive->addFile( $manifest_filename , "META-INF/manifest.xml" );

#   create a temporary file for styles,
#   create the styles.xml file, and
#   add it to the archive.
#
$styles_filename = File::Spec->catfile(File::Spec->tmpdir(),
"styles.xml");
open STYLESFILE, ">$styles_filename" or
    die("Cannot open $styles_filename");

create_styles_file( );

close STYLESFILE;
$archive->addFile( $styles_filename , "styles.xml" );
```

The manifest file's contents depend upon the pictures, so here's the code for creating the manifest file:

```
sub create_manifest_file
{
    print MANIFESTFILE << "MANIFEST";
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE manifest:manifest
    PUBLIC "-//OpenOffice.org//DTD Manifest 1.0//EN" "Manifest.dtd">
<manifest:manifest
    xmlns:manifest="urn:oasis:names:tc:opendocument:xmlns:manifest:►
1.0">
    <manifest:file-entry
        manifest:media-
type="application/vnd.oasis.opendocument.presentation"
        manifest:full-path="/"/>
    <manifest:file-entry manifest:media-type="text/xml"
        manifest:full-path="content.xml"/>
    <manifest:file-entry manifest:media-type="text/xml"
        manifest:full-path="styles.xml"/>
    <manifest:file-entry manifest:media-type="text/xml"
        manifest:full-path="meta.xml"/>
    <manifest:file-entry manifest:media-type=""
        manifest:full-path="Pictures/"/>
MANIFEST
    for ($i=0; $i < @piclist; $i += 2)
    {
        print MANIFESTFILE
            qq!<manifest:file-entry manifest:media-type="image/jpeg"!,
            qq! manifest:full-path="Pictures/$piclist[$i]"/>\n!;
    }
    print MANIFESTFILE "</manifest:manifest>\n";
}
```

We now create the main part of the content file.

```
#   create a temporary file for content
#
$content_filename = File::Spec->catfile(File::Spec->tmpdir(),
    "content.xml");
open CONTENTFILE, ">$content_filename" or
    die("Cannot open $content_filename");
insert_header(); ❶
```

```
for ($i=0; $i < @piclist; $i += 2)
{
    $slide_number = ($i / 2) + 1;
    ($file_name, $description) = @piclist[$i,$i+1];

    ($width, $height) = get_jpg_dimensions($file_name);

    # Presume that pictures are 72 dpi; convert width & height
    # to centimeters.

    $width = ($width / 72) * 2.54;
    $height = ($height / 72) * 2.54;
    $left_edge = (27.94-$width) / 2; ❷

    $archive->addFile( $file_name, "Pictures/$file_name");
    print CONTENTFILE << "ONE_PAGE";

<draw:page draw:name="page$slide_number"
 draw:master-page-name="PM1" draw:style-name="dp7"
 presentation:presentation-page-layout-name="slidepage">
    <draw:frame draw:layer="layout" presentation:class="title"
        presentation:style-name="standard"
        svg:x="4.5cm" svg:y="2.25cm"
        svg:width="20cm" svg:height="2cm">
        <draw:text-box>
            <text:p text:style-name="P1">$description</text:p>
        </draw:text-box>
    </draw:frame>
    <draw:frame draw:layer="layout"
      svg:width="${width}cm" svg:height="${height}cm"
      svg:x="${left_edge}cm" svg:y="4cm">
        <draw:image
            xlink:href="Pictures/$file_name"
            xlink:type="simple" xlink:show="embed"
            xlink:actuate="onLoad"/>
    </draw:frame>
ONE_PAGE
    if ($slide_number != 1)
    {
        insert_back_button();
    }
    if ($slide_number != int(@piclist/2))
    {
        insert_next_button();
    }
    print CONTENTFILE "</draw:page>\n";

}

insert_footer(); ❸

close CONTENTFILE;

$archive->addFile( $content_filename , "content.xml" );
$archive->overwriteAs( $ARGV[1] );

unlink $styles_filename; ❹
unlink $content_filename;
unlink $manifest_filename;
```

❶ This subroutine provides the boilerplate for the root element, namespaces, and `<office:automatic-styles>` for the content file.

❷ The `$left_edge` variable lets us center the picture horizontally on the slide.

❸ This subroutine provides the boilerplate for the closing tags in the content section.

❹ This deletes the temporary files once the OpenDocument file is complete.

The code to insert the next and back buttons shows the use of interaction:

```
sub insert_back_button
{
    print CONTENTFILE << "BACK_BUTTON";
<draw:rect draw:layer="layout" draw:text-style-name="centered"
 svg:x="2.2cm" svg:y="18.3cm"
 svg:width="2cm" svg:height="1cm"
 draw:style-name="buttonborderstyle">
    <office:event-listeners>
        <presentation:event-listener script:event-name="dom:click"
         presentation:action="previous-page"/>
    </office:event-listeners>
    <text:p text:style-name="P1">
        <text:span text:style-name="buttontext">Back</text:span>
    </text:p>
</draw:rect>
BACK_BUTTON
}

sub insert_next_button
{
    print CONTENTFILE << "NEXT_BUTTON";
<draw:rect draw:layer="layout"
 svg:x="24cm" svg:y="18.3cm"
 svg:width="2cm" svg:height="1cm"
 draw:style-name="buttonborderstyle">
    <office:event-listeners>
        <presentation:event-listener script:event-name="dom:click"
         presentation:action="next-page"/>
    </office:event-listeners>
    <text:p text:style-name="P1">
        <text:span text:style-name="buttontext">Next</text:span>
    </text:p>
</draw:rect>
NEXT_BUTTON
}
```

Here's the code that inserts the heading material into the `content.xml` file. Note the `presentation:transition-style` in the `<style:style>` named `dp1`.

```
sub insert_header
{
    print CONTENTFILE <<"HEADER";
<?xml version="1.0" encoding="UTF-8"?>
<office:document-content
    xmlns:office="urn:oasis:names:tc:opendocument:xmlns:office:1.0"
    xmlns:style="urn:oasis:names:tc:opendocument:xmlns:style:1.0"
    xmlns:text="urn:oasis:names:tc:opendocument:xmlns:text:1.0"
    xmlns:table="urn:oasis:names:tc:opendocument:xmlns:table:1.0"
    xmlns:draw="urn:oasis:names:tc:opendocument:xmlns:drawing:1.0"
    xmlns:fo="urn:oasis:names:tc:opendocument:xmlns:►
xsl-fo-compatible:1.0"
    xmlns:xlink="http://www.w3.org/1999/xlink"
    xmlns:number="urn:oasis:names:tc:opendocument:xmlns:datastyle:1.0"
    xmlns:presentation="urn:oasis:names:tc:opendocument:xmlns:►
presentation:1.0"
    xmlns:svg="urn:oasis:names:tc:opendocument:xmlns:►
svg-compatible:1.0"
    xmlns:form="urn:oasis:names:tc:opendocument:xmlns:form:1.0"
    xmlns:script="urn:oasis:names:tc:opendocument:xmlns:script:1.0"
    xmlns:dom="http://www.w3.org/2001/xml-events"
    office:version="1.0">
 <office:scripts/>
 <office:automatic-styles>
    <style:style style:name="dp1" style:family="drawing-page">
        <style:properties
         presentation:transition-speed="medium"
         presentation:background-visible="true"
         presentation:background-objects-visible="true"
         smil:type="barWipe"
         smil:subtype="leftToRight"/>
    </style:style>
    <style:style style:name="P1" style:family="paragraph">
        <style:paragraph-properties fo:text-align="center"/>
        <style:text-properties fo:font-size="18pt"/>
    </style:style>
    <style:style style:name="buttonborderstyle"
       style:family="graphic">
        <style:graphic-properties
            draw:stroke="solid"
            svg:stroke-width="0.05cm"
            svg:stroke-color="#cccccc"
            draw:fill="solid" draw:fill-color="#ffffcc"/>
    </style:style>
    <style:style style:name="buttontext" style:family="text">
        <style:text-properties
            fo:font-family="Helvetica"
            style:font-family-generic="swiss"
            fo:font-size="10pt"/>
    </style:style>
 </office:automatic-styles>
 <office:body>
    <office:presentation>
HEADER
}

sub insert_footer
{
    print CONTENTFILE <<"FOOTER";
    </office:presentation>
 </office:body>
</office:document-content>
FOOTER
}
```

Finally, we have the code to create the information in the `styles.xml` file. It is pure boilerplate, adapted from an existing presentation file, with lots of unnecessary information removed.

```
sub create_styles_file
{
    print STYLESFILE << "STYLES";
<office:document-styles
 xmlns:office="urn:oasis:names:tc:opendocument:xmlns:office:1.0"
 xmlns:style="urn:oasis:names:tc:opendocument:xmlns:style:1.0"
 xmlns:text="urn:oasis:names:tc:opendocument:xmlns:text:1.0"
 xmlns:table="urn:oasis:names:tc:opendocument:xmlns:table:1.0"
 xmlns:draw="urn:oasis:names:tc:opendocument:xmlns:drawing:1.0"
 xmlns:fo="urn:oasis:names:tc:opendocument:xmlns:►
xsl-fo-compatible:1.0"
 xmlns:xlink="http://www.w3.org/1999/xlink"
 xmlns:number="urn:oasis:names:tc:opendocument:xmlns:datastyle:1.0"
 xmlns:presentation="urn:oasis:names:tc:opendocument:xmlns:►
presentation:1.0"
 xmlns:svg="urn:oasis:names:tc:opendocument:xmlns:svg-compatible:1.0"
 xmlns:dom="http://www.w3.org/2001/xml-events"
 office:version="1.0">

<office:styles>
    <style:presentation-page-layout style:name="slidepage">
        <presentation:placeholder presentation:object="title"
          svg:x="4.5cm" svg:y="2.25cm"
          svg:width="20cm" svg:height="2cm"/>
        <presentation:placeholder presentation:object="graphic"
          svg:x="2cm" svg:y="5.5cm"
          svg:width="20cm" svg:height="13cm"/>
    </style:presentation-page-layout>
</office:styles>

<office:automatic-styles>
    <style:page-master style:name="PM1">
        <style:properties
         fo:margin-top="1cm" fo:margin-bottom="1cm"
         fo:margin-left="1cm" fo:margin-right="1cm"
         fo:page-width="27.94cm" fo:page-height="21.59cm"
         style:print-orientation="landscape"/>
    </style:page-master>
</office:automatic-styles>

<office:master-styles>
    <draw:layer-set>
        <draw:layer draw:name="layout"/>
        <draw:layer draw:name="background"/>
        <draw:layer draw:name="backgroundobjects"/>
        <draw:layer draw:name="controls"/>
        <draw:layer draw:name="measurelines"/>
    </draw:layer-set>

    <style:master-page style:name="Default" style:page-master-
name="PM1"
     draw:style-name="dp1">
    </style:master-page>
</office:master-styles>

</office:document-styles>
STYLES
}
```

We are omitting the `get_jpg_dimensions` code that gets the width and height of a JPEG; it is not relevant to creating the OpenDocument file. If you are interested in seeing it, just download it from the book's web site.

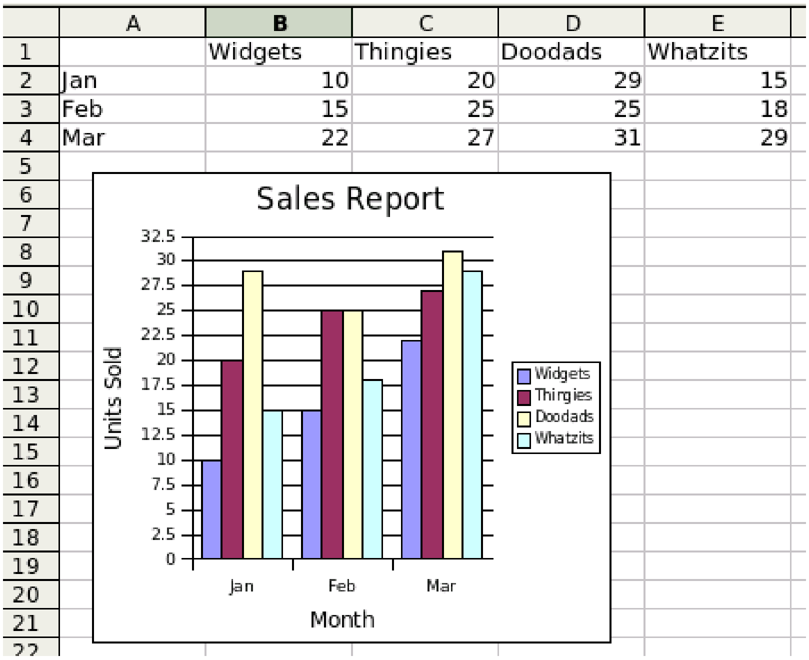
Chapter 8. Charts

OpenDocument includes a powerful specification for making charts. You may place a chart in a word processing document, a drawing, a presentation, or a spreadsheet.

Chart Terminology

Before proceeding to the actual XML, let's define some of the terms that are used when talking about charts. Consider the spreadsheet section shown in Figure 8.1, "Chart Derived from Spreadsheet". In this chart, there are three categories: January, February, and March–the months of the first quarter of the year. Each of the items sold has a series of three data points or values. The values are the numbers in the spreadsheet, and they appear on the y-axis.

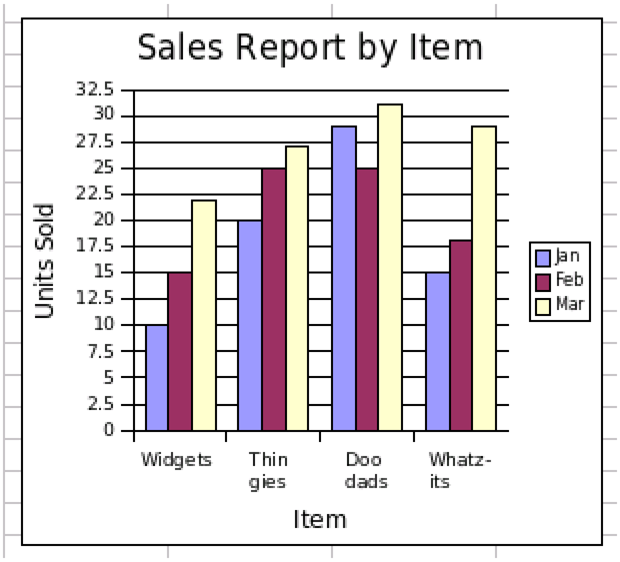
Figure 8.1. Chart Derived from Spreadsheet



	A	B	C	D	E
1		Widgets	Thingies	Doodads	Whatzits
2	Jan	10	20	29	15
3	Feb	15	25	25	18
4	Mar	22	27	31	29

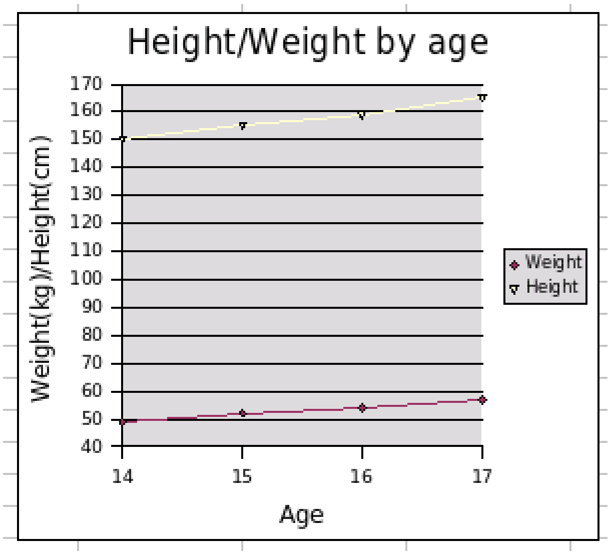
It is also possible to describe a chart where each row is a series of data points, and the columns are the categories. This produces a chart of sales by items sold, shown in Figure 8.2, "Chart with Series in Rows". Here, each month has a series of four data points, one for each item.

Figure 8.2. Chart with Series in Rows



In an *x*-*y* chart (such as the scatter chart in Figure 8.3, "Scatter Chart"), the *x*-axis is called the *domain*, as in the domain of an algebraic function.

Figure 8.3. Scatter Chart



Charts are Objects

Rather than being inserted directly into the `content.xml`, the chart is inserted as a `<draw:object>` element. This element will have an `xlink:href` attribute of the form `./Object 1`. If you know your URLs, you see that this points to a subdirectory named `Object 1`. Inside that directory you will find another `content.xml` file that contains the chart data and style information.

Note

The object's directory also contains a `styles.xml` file, but it is just a placeholder; Its root element is a `<office:document-styles>` that contains an empty `<office:styles>` element. If you are creating a chart from scratch, you may omit the `styles.xml` file.

In addition to creating the link and the subdirectory, you must also add entries in the `META-INF/manifest.xml` file in order for an OpenDocument-compatible application to locate the chart. The entries must define the paths for the subdirectory and the `content.xml` and `style.xml` files in the subdirectory (if present).

Common Attributes for <draw:object>

No matter which type of document you insert a chart into, the chart's `<draw:object>` element will contain the following attributes:

- `xlink:href` contains the URL of the object subdirectory.
- `xlink:type` has the value `simple`.
- `xlink:show` has the value `embed`.
- `xlink:actuate` has the value `onLoad`.

Charts in Word Processing Documents

The `<draw:object>` element for a chart embedded in a word processing document will be contained in a `<draw:frame>`.

Since the object is inside a frame, the `<draw:frame>` element will contain all the attributes described in the section called "Body Information for Frames" with one exception. Instead of an `fo:min-height` attribute, the object will have a `svg:height` attribute to specify the height of the chart. The `<draw:object>` will also contain a `draw:notify-on-update-of-ranges` attribute whose value is the name of the table that is being charted. Example 8.1, "XML for a Chart in a Word Processing Document" shows the XML for a chart anchored as a paragraph.

Example 8.1. XML for a Chart in a Word Processing Document

```
<text:p text:style-name="Text_20_body">
   <draw:frame draw:style-name="fr1" draw:name="Object1"
     text:anchor-type="paragraph"
     svg:width="7.999cm" svg:height="7.001cm"
     draw:z-index="0">
       <draw:object xlink:href="./Object 1"
         xlink:type="simple" xlink:show="embed"
         xlink:actuate="onLoad"
         draw:notify-on-update-of-ranges="Table1"/>
         <draw:image xlink:href="./ObjectReplacements/Object 1"
           xlink:type="simple" xlink:show="embed"
           xlink:actuate="onLoad"/>
   </draw:frame>
</text:p>
```

The `<draw:image>` element contains a reference to some sort of vector-based representation of the chart. In case the chart object itself is missing, an OpenDocument-compatible application can display this replacement picture.

Charts in Drawings

When you insert a chart into a drawing, the `<draw:object>` again goes inside a `<draw:frame>` element which gets these attributes:

- `draw:layer` (either `layout` or `background`).
- `svg:x` and `svg:y` determine the upper left corner of the chart.
- `svg:width` and `svg:height` determine the chart’s dimensions.
- `draw:style-name` gives the controlling style for the chart; usually `standard`.

The `<draw:object>` element has the same attributes as described in the section called “Common Attributes for `<draw:object>`”.

Charts in Spreadsheets

Charts in spreadsheets are special; they display data that is within the rows and columns of the spreadsheet. Here are the `<draw:object>` attributes in question:

`draw:notify-on-update-of-ranges`
: The value of this attribute is the range of cells that contain the chart data, such as `Sheet1.A1:Sheet1.D4`.

`table:end-cell-address`
: This is the cell that the lower right corner of the chart covers, in a form such as `Sheet1.G21`. (The starting cell is the table cell that holds the `<draw:object>` element.)

`table:end-x`, `table:end-y`
: These attributes have a *length* value that tells how far the chart extends into the ending cell. If these attributes are not present, the chart will not display. You may set the values to zero.

`svg:x`, `svg:y`
: These attributes have a *length* value that tells how far the upper left corner of the chart is from the upper left of the first cell in which the chart resides. The default value for these attributes is zero.

`svg:width`, `svg:height`
: These attributes give the size of the chart.

Example 8.2, “XML for Chart in Spreadsheet” shows the XML that embeds a chart shown in Figure 8.1, “Chart Derived from Spreadsheet” into a spreadsheet.

Example 8.2. XML for Chart in Spreadsheet

```
<draw:frame
  table:end-cell-address="Sheet1.D22"
  table:end-x="0.6728in" table:end-y="0.0213in"
  draw:z-index="0"
  svg:width="3.1441in" svg:height="2.752in"
  svg:x="0.1953in" svg:y="0.115in">
    <draw:object
      draw:notify-on-update-of-ranges="Sheet1.A1:Sheet1.E4"
      xlink:href="./Object 1" xlink:type="simple"
      xlink:show="embed" xlink:actuate="onLoad"/>
    <draw:image
      xlink:href="./ObjectReplacements/Object 1
      xlink:type="simple" xlink:show="embed"
      xlink:actuate="onLoad"/>
</draw:frame>
```

Chart Contents

You find the actual chart data and specifications in the `content.xml` file that is in the object subdirectory. This file follows the same general pattern that we have seen for content files of all the other document types. The first child of the root `<office:document-content>` is an `<office:automatic-styles>` element that contains all the styles to control the chart’s presentation.

The styles are followed by the `<office:body>`, which contains an `<office:chart>` element which in turn contains a `<chart:chart>` element. This has child elements that specify:

- The chart title, subtitle, and legend.
- The plot area, which includes dimensions and:
    - The chart axes and grid.
    - Chart categories and data series.
- A `<table:table>` that provides the data to be charted.

Now let's take a closer look at the `chart:chart` element and its attributes and children. The `chart:class` attribute tells what kind of chart to draw: `chart:line`, `chart:area` (stacked areas), `chart:circle` (pie chart), `chart:ring`, `chart:scatter`, `chart:radar` (called "net") in OpenOffice.org, `chart:bar`, `chart:stock`, and `chart:add-in`.

The `<chart:chart>` element has these children, in this order:

- An optional `<chart:title>` element.
- An optional `<chart:subtitle>` element.
- An optional `<chart:legend>` element.
- A `<chart:plot-area>` element that describes the axes and grid.
- An optional `<table:table>` containing the table data.

The `<chart:title>` and `<chart:subtitle>` elements have `svg:x` and `svg:y` attributes for positioning, and a `chart:style-name` for presentation. They contain a `<text:p>` element that gives the title (or subtitle) text, as shown in Example 8.3, "Example of Chart Title and Subtitle"

Example 8.3. Example of Chart Title and Subtitle

```
<chart:title svg:x="2.225cm" svg:y="0.28cm" chart:style-name="ch2">
    <text:p>Sales Report</text:p>
</chart:title>

<chart:subtitle svg:x="4.716cm" svg:y="0.814cm"
  chart:style-name="ch3">
    <text:p>First Quarter</text:p>
</chart:subtitle>
```

The `<chart:legend>` element has a `chart:legend-position` attribute that gives the relative location of the legend; `top`, `start` (left), `bottom`, or `end` (right), and an absolute `svg:x` and `svg:y` position. It also has a `chart:style-name` attribute to determine the presentation of the text in the legend.

The Plot Area

The next element in line is a `<chart:plot-area>` element. This element is where the action is. It establishes the location of the chart with the typical `svg:x`, `svg:y`, `svg:width`, and `svg:height` attributes.

If you are creating a chart from a spreadsheet, you will specify the source of the data in the `table:cell-range-address` attribute. Depending on whether this range of cells contains labels for the rows or columns, you must set `chart:data-source-has-labels` to `none`, `row`, `column`, or `both`. The `<chart:table-number-list>` is not used in the XML format, and should be set to `0`.

You may be tempted to overlook the standard `chart:style-name` attribute, but that would be a mistake, because that style is just *packed* with information.

`chart:lines`
: `true` for a line chart, `false` for any other type of chart.

`chart:symbol-type`
: Used only with line charts, this is set to `automatic` to allow the application to cycle through a series of pre-defined symbols to mark points on the line chart.

`chart:splines`, `chart:spline-order`, `chart:spline-resolution`
: If you are using splines instead of lines, then `chart:interpolation` will be `cubic-spline`, and you must specify the `chart:spline-order` (2 for cubic splines). The `chart:spline-resolution` tells how smooth the curve is; the larger the number, the smoother the curve; the default value is 20.

`chart:vertical`, `chart:stacked`, `chart:percentage`, `chart:connect-bars`
: These booleans are used for bar charts. If `chart:vertical` is `true` then bars are drawn along the vertical axis from left to right (the default is `false` for bars drawn up and down along the horizontal axis). `chart:stacked` tells whether bars are stacked or side-by-side. This is mutually exclusive with `chart:percentage`, which draws stacked bars by default. The `chart:connect-bars` attribute is only used for stacked bars or percentage charts; it draws lines connecting the various levels of bars.

`chart:lines-used`
: The default value is zero; it is set to one if a bar chart has lines on it as well.

`chart:stock-updown-bars`, `chart:stock-with-volume`, `chart:japanese-candle-stick`
: These boolean attributes apply only when `chart:class` is `stock`. The `chart:japanese-candle-stick` attribute is set to `true` if you want a box drawn between the upper and lower limits rather than just two lines.

`chart:series-source`
: If your source data has its data series in rows instead of columns, set this attribute to `rows` instead of the default `columns`.

`chart:data-label-number`
: Is the data labeled with the `value`, a `percentage`, or `none` (the default).

`chart:data-label-text`, `chart:data-label-symbol`
: Should all data points have a text label (the name of the corresponding series) and/or the legend symbol next to them? Set these to `true` or the default `false`.

Example 8.4, “Plot Area and Style” shows the opening `<chart:plot-area>` element (and its associated style) for the bar chart in Figure 8.1, “Chart Derived from Spreadsheet”.

Example 8.4. Plot Area and Style

```
<chart:plot-area
   chart:style-name="ch5"
   table:cell-range-address="Sheet1.$A$1:.$E$4"
   chart:data-source-has-labels="both"
   chart:table-number-list="0"
   svg:x="0.158cm" svg:y="1.672cm"
   svg:width="6.006cm" svg:height="5.181cm">

<!-- the associated style -->

<style:style style:name="ch5" style:family="chart">
   <style:chart-properties
      chart:series-source="columns" ❶
      chart:vertical="false"
      chart:connect-bars="false"
      chart:lines="false" chart:lines-used="0" ❷
      chart:symbol-type="none"
      chart:data-label-number="none" ❸
      chart:data-label-text="false"
      chart:data-label-symbol="false"
      chart:interpolation="none"
      chart:mean-value="false" ❹
      chart:error-margin="0" chart:error-lower-limit="0"
      chart:error-upper-limit="0" chart:error-category="none"
      chart:error-percentage="0" chart:regression-type="none"
      chart:three-dimensional="false" chart:deep="false" ❺
      chart:stock-with-volume="false"
      chart:japanese-candle-stick="false"/>
</style:style>
```

❶ The values for these first three attributes are the default values, so they aren't really needed in this case.

❷ If you were creating a line chart, you'd need the first of these attributes, but you can leave them out for a bar chart. The second attribute is not in the OpenDocument specification, but does appear in the OpenOffice.org document.

❸ These are all set to `none` or `false` so that no extra labelling appears next to the data points.

❹ Because this is an "essentials" book, we didn't talk about these attributes at all. They are used if you use the Insert/Statistics menu in OpenOffice.org.

❺ Finally, these attributes are all `false` because this is neither a three-d chart nor a stock chart.

Chart Axes and Grid

Within the `<chart:plot-area>` element are two `<chart:axis>` elements; the first for the *x*-axis and the second for the *y*-axis. For pie charts, there is only one axis; the *y*-axis.

Each `<chart:axis>` has a `chart:name` attribute, which is either `primary-x` or `primary-y`. The `chart:class` attribute tells whether the axis represents a `category`, `value`, or `domain`. (This last is for the *x*-axis of a scatter chart.) There is a child `chart:categories` if this axis determines the categories. Of course, there's a `chart:style-name`, and the style it refers to also contains oodles of information about how to display the axis:

`chart:display-label`
: A boolean that determines whether to display a label with this axis or not.

`chart:tick-marks-major-inner`, `chart:tick-marks-major-outer`, `chart:tick-marks-minor-inner`, `chart:tick-marks-minor-outer`
: These four booleans tell whether you want tick marks at major and minor intervals, and whether you want them to appear outside the chart area or inside the chart area.

`chart:logarithmic`
: Set this to `true` if you want a logarithmic scale for the numbers on the given axis.

`text:line-break`
: In order to fit labels into small charts, the application will break words. For example, a category label of "Northwest" may appear with "North" on one line and "west" beneath it. You can turn off this action by setting the attribute to `false`.

`chart:text-overlap`
: If you turn off line break and your chart is small, but its labels are long, then the labels may overlap. If you don't want this to happen, set this attribute to its default value of `false`. An application will then avoid displaying some of the labels rather than have labels display on top of one another. If you don't mind the overwriting, set this attribute to `true`.

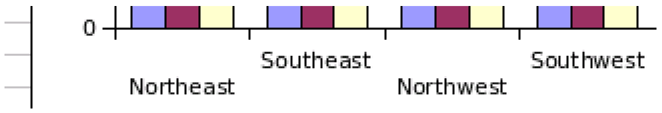
`chart:label-arrangement`
: Ordinarily the labels on a chart appear `side-by-side` (the default value). You may avoid overlap by setting this value to `stagger-even` or `stagger-odd`. Figure 8.4, "Chart With Even-Staggered Labels" shows the labels for a chart with this attribute set to `stagger-even`.

`chart:visible`
: Set this to `false` if you don't want to see any labels or tick marks at all.

Warning

Don't set this to `false` unless you have a compelling reason to do so. Graphs without labels are confusing at best and misleading or useless at worst.

Figure 8.4. Chart With Even-Staggered Labels



If your axis has a title, then the `<chart:axis>` element will have a `<chart:title>` child element, formatted exactly like the chart's main title.

The last child of the `<chart:axis>` element is the optional `<chart:grid>` element. Its `chart:class` attribute tells whether you want grid lines at major intervals only (`major`), or at both major and minor intervals (`minor`). For no grid lines, omit the element.

Data Series

We still haven't finished the `<chart:plot-area>` element yet; after specifying the axes and grid, we must now define what data series are in the chart.

The XML will continue with one `<chart:series>` element for each data series in the chart. It has a `chart:style-name` that refers to a style for that data series.

For line charts, this style needs to specify only the `draw:fill-color` and `svg:stroke-color`. For bar and pie charts, you need to specify only `draw:fill-color`.

For line and bar charts, each `<chart:series>` element contains a `<chart:data-point>` element; its `chart:repeated` attribute tells how many data points are in the series. A pie chart has only one `chart:series` element that contains multiple `chart:data-point` elements; one for each pie slice, and each will have its own `chart:style-name` attribute.

Wall and Floor

The chart wall is the area bounded by the axes (as opposed to the plot area, which is the entire chart). The empty `<chart:wall>` element has a `chart:style-name` attribute, used primarily to set the background color. The chart floor is applicable only to three-dimensional charts, and will be covered in that section.

This has been an immense amount of explanation, and we need to see how this all fits together. Example 8.5, "Styles and Content for a Bar Chart" shows the XML (so far) for the chart shown in Figure 8.1, "Chart Derived from Spreadsheet".

Example 8.5. Styles and Content for a Bar Chart

```
<chart:chart svg:width="8.002cm" svg:height="6.991cm"
  chart:class="chart:bar" chart:style-name="ch1">
    <chart:title svg:x="2.549cm" svg:y="0.138cm"
      chart:style-name="ch2">
        <text:p>Sales Report</text:p>
    </chart:title>
    <chart:legend chart:legend-position="end"
      svg:x="6.492cm" svg:y="2.824cm" chart:style-name="ch3"/>

    <chart:plot-area chart:style-name="ch4"
      table:cell-range-address="Sheet1.$A$1:.$E$4"
      chart:data-source-has-labels="both"
      chart:table-number-list="0"
      svg:x="0.16cm" svg:y="0.945cm"
      svg:width="6.013cm" svg:height="5.908cm">
        <chart:axis chart:dimension="x"
          chart:name="primary-x" chart:style-name="ch5">
            <chart:categories
                table:cell-range-address="local-table.A2:.A4"/>
        </chart:axis>

        <chart:axis chart:dimension="y"
          chart:name="primary-y" chart:style-name="ch6">
            <chart:grid chart:class="major"/>
        </chart:axis>

        <chart:series chart:style-name="ch7">
            <chart:data-point chart:repeated="3"/>
        </chart:series>
        <chart:series chart:style-name="ch8">
            <chart:data-point chart:repeated="3"/>
        </chart:series>
        <chart:series chart:style-name="ch9">
            <chart:data-point chart:repeated="3"/>
        </chart:series>
        <chart:series chart:style-name="ch10">
            <chart:data-point chart:repeated="3"/>
        </chart:series>

        <chart:wall chart:style-name="ch11"/>

        <chart:floor chart:style-name="ch12"/>
    </chart:plot-area>

    <!-- data table follows -->

</chart:chart>
```

Example 8.6, “Styles for Bar Chart Excerpt” shows the corresponding styles, cut down to minimal size. For example, in the styles for `<chart:series>` elements, we have left out the `<style:text-properties>` element because the bars are not labelled. For variety, we have used `fo:font-family` on some styles to explicitly specify a font, and in others we have used `style:font-family-generic` to specify the font. Comments have been added to indicate which styles apply to which parts of the chart.

Example 8.6. Styles for Bar Chart Excerpt

```
<!-- style for <chart:chart> element -->
<style:style style:name="ch1" style:family="chart">
   <style:graphic-properties draw:stroke="solid"
     draw:fill-color="#ffffff"/>
</style:style>

<!-- style for <chart:title> element -->
<style:style style:name="ch2" style:family="chart">
   <style:text-properties
     fo:font-family="&apos;Bitstream Vera Sans&apos;"
     style:font-family-generic="swiss" fo:font-size="13pt"/>
</style:style>

<!-- style for <chart:legend> element -->
<style:style style:name="ch3" style:family="chart">
    <style:properties style:font-family-generic="swiss"
      fo:font-size="6pt"/>
</style:style>

<!-- style for <chart:plot-area> element -->
<style:style style:name="ch4" style:family="chart">
    <style:chart-properties chart:series-source="columns"
      chart:lines="false" chart:vertical="false"
      chart:connect-bars="false"/>
</style:style>

<!-- style for first <chart:axis> (x-axis) -->
<style:style style:name="ch5" style:family="chart"
  style:data-style-name="N0">
    <style:chart-properties chart:display-label="true"
      chart:tick-marks-major-inner="false"
      chart:tick-marks-major-outer="true"
      text:line-break="true"
      chart:label-arrangement="side-by-side" chart:visible="true"/>
    <style:graphic-properties draw:stroke="solid"
      svg:stroke-width="0cm" svg:stroke-color="#000000"/>
    <style:text-properties style:font-family-generic="swiss"
      fo:font-size="7pt"/>
</style:style>

<!-- style for second <chart:axis> (y-axis) -->
<style:style style:name="ch6" style:family="chart"
  style:data-style-name="N0">
    <style:chart-properties chart:display-label="true"
      chart:tick-marks-major-inner="false"
      chart:tick-marks-major-outer="true"
      text:line-break="false"\
      chart:label-arrangement="side-by-side" chart:visible="true"/>
    <style:graphic-properties draw:stroke="solid"
      svg:stroke-width="0cm" svg:stroke-color="#000000"/>
    <style:text-properties style:font-family-generic="swiss"
      fo:font-size="7pt"/>
</style:style>

<!-- style for the first <chart:series> element -->
<style:style style:name="ch7" style:family="chart">
    <style:graphic-properties draw:fill-color="#9999ff"/>
</style:style>
```

```
<!-- style for the second <chart:series> element -->
<style:style style:name="ch8" style:family="chart">
    <style:graphic-properties draw:fill-color="#993366"/>
</style:style>

<!-- style for the third <chart:series> element -->
<style:style style:name="ch9" style:family="chart">
    <style:graphic-properties draw:fill-color="#ffffcc"/>
</style:style>

<!-- style for the fourth <chart:series> element -->
<style:style style:name="ch10" style:family="chart">
    <style:graphic-properties draw:fill-color="#ccffff"/>
</style:style>

<!-- style for the <chart:wall> element -->
<style:style style:name="ch11" style:family="chart">
    <style:graphic-properties draw:stroke="none" draw:fill="none"/>
</style:style>

<!-- style for the <chart:floor> element -->
<style:style style:name="ch12" style:family="chart">
    <style:graphic-properties draw:stroke="none" draw:fill-
color="#999999"/>
</style:style>
```

The Chart Data Table

Following the plot area is a table containing the data to be displayed. Even if you are creating a chart from a spreadsheet, OpenOffice.org does *not* look at the spreadsheet cells for the data—it looks at the internal table in the chart object's `content.xml` file.

Compared to the chart and plot area definitions, the data table is positively anticlimactic. The `<table:table>` element has a `table:name` attribute which is set to `local-table`.

The first child of the `<table:table>` is a `<table:table-header-columns>` element that contains an empty `<table:table-column>` element. This is followed by a `<table:table-header-rows>` element that contains the first row of the table. Finally, a `<table:table-rows>` element contains the remaining data, one `<table:table-row>` at a time.

Example 8.7, "Table for Bar Chart" gives an excerpt of the table that was used in Figure 8.1, "Chart Derived from Spreadsheet".

Example 8.7. Table for Bar Chart

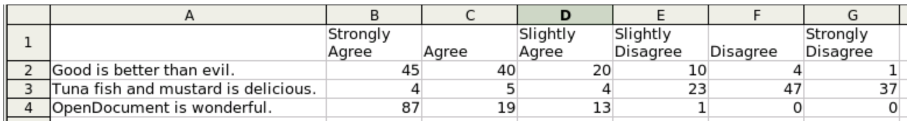
```
<table:table table:name="local-table">
    <table:table-header-columns>
        <table:table-column/>
    </table:table-header-columns>
    <table:table-columns>
        <table:table-column table:number-columns-repeated="4"/>
    </table:table-columns>

    <table:table-header-rows>
        <table:table-row>
            <table:table-cell>
                <text:p/>
            </table:table-cell>
            <table:table-cell table:value-type="string">
                <text:p>Widgets</text:p>
            </table:table-cell>
            <table:table-cell table:value-type="string">
                <text:p>Thingies</text:p>
            </table:table-cell>
            <table:table-cell table:value-type="string">
                <text:p>Doodads</text:p>
            </table:table-cell>
            <table:table-cell table:value-type="string">
                <text:p>Whatzits</text:p>
            </table:table-cell>
        </table:table-row>
    </table:table-header-rows>

    <table:table-rows>
        <table:table-row>
            <table:table-cell table:value-type="string">
                <text:p>Jan</text:p>
            </table:table-cell>
            <table:table-cell
              table:value-type="float" table:value="10">
                <text:p>10</text:p>
            </table:table-cell>
            <table:table-cell
              table:value-type="float" table:value="20">
                <text:p>20</text:p>
            </table:table-cell>
            <table:table-cell
              table:value-type="float" table:value="29">
                <text:p>29</text:p>
            </table:table-cell>
            <table:table-cell
              table:value-type="float" table:value="15">
                <text:p>15</text:p>
            </table:table-cell>
        </table:table-row>

        <!-- February row, similar to January above -->

        <table:table-row>
            <table:table-cell table:value-type="string">
                <text:p>Mar</text:p>
            </table:table-cell>
            <table:table-cell
              table:value-type="float" table:value="22">
                <text:p>22</text:p>
            </table:table-cell>
            <table:table-cell
              table:value-type="float" table:value="27">
                <text:p>27</text:p>
            </table:table-cell>
            <table:table-cell
```

```
                table:value-type="float" table:value="31">
                  <text:p>31</text:p>
              </table:table-cell>
              <table:table-cell
                 table:value-type="float" table:value="29">
                  <text:p>29</text:p>
              </table:table-cell>
          </table:table-row>
      </table:table-rows>
</table:table>
```

Case Study - Creating Pie Charts

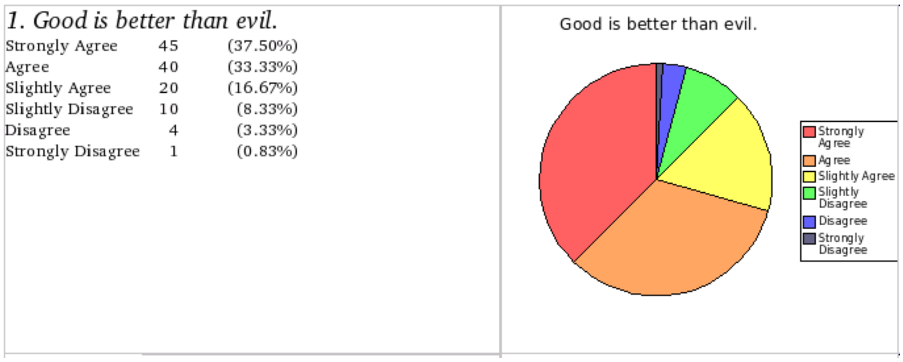
We are now prepared to do a rather complex case study. We will begin with an OpenDocument spreadsheet that contains the results of a survey[13], as shown in Figure 8.5, "Spreadsheet with Survey Responses". Our goal is to create a word processing document. Each question will be displayed in a two-column section. The left column will contain the question and the results in text form; the right column will contain a pie chart of the responses to the question. The result will look like Figure 8.6, "Text Document with Survey Responses".

Figure 8.5. Spreadsheet with Survey Responses



	A	B	C	D	E	F	G
1		Strongly Agree	Agree	Slightly Agree	Slightly Disagree	Disagree	Strongly Disagree
2	Good is better than evil.	45	40	20	10	4	1
3	Tuna fish and mustard is delicious.	4	5	4	23	47	37
4	OpenDocument is wonderful.	87	19	13	1	0	0

Figure 8.6. Text Document with Survey Responses



1. Good is better than evil.

Strongly Agree	45	(37.50%)
Agree	40	(33.33%)
Slightly Agree	20	(16.67%)
Slightly Disagree	10	(8.33%)
Disagree	4	(3.33%)
Strongly Disagree	1	(0.83%)

[13] This survey uses what is called a six-point Likert scale. If you are setting up a survey, always make sure you have an even number of choices. If you have an odd number of choices with "Neutral" in the middle, people will head for the center like moths to a flame. Using an even number of choices forces respondents to make a decision.

The Perl code is fairly lengthy, though most of it is just “boilerplate.” We have broken it into sections for ease of analysis. We will use the `XML::DOM` module to parse the input file for use with the Document Object Model. We won’t use the DOM to create the output file; we’ll just create raw XML text and put it into temporary files, which will eventually be added to the output .zip file. Let’s begin with the variable declarations. [You will find the entire Perl program in file `chartmaker.pl` in directory `ch08` in the downloadable example files.]

```
#!/usr/bin/perl

use Archive::Zip;
use XML::DOM;
use warnings;
use strict;

#
#   Command line arguments:
#       input file name
#       output file name

my $doc;        # the DOM document
my $rows;       # all the <table:table-row> elements
my $n_rows;     # number of rows
my $row;        # current row number
my $col;        # current column number
my @data;       # contents of current row
my $sum;        # sum of the row items
my @legends;    # legends for the graph

my $main_handle;        # content/style file handle ❶
my $main_filename;      # content/style file name

my $manifest_handle;    # manifest file handle
my $manifest_filename;  # manifest file name

my $chart_handle;       # chart file handle
my $chart_filename;     # chart file name

my @temp_filename;      # list of all temporary filenames created ❷
my $item;               # foreach loop variable

my $zip;                # output zip file name

my $percent;    # string holding nicely formatted percent value
```

❶ The `$`*`name`*`_handle` and `$`*`name`*`_filename` are the file handle and file name returned by `Archive::Zip->tempFile()`.

❷ All the temporary files need to be kept around until the .zip file is finally written; adding a file to the archive just adds the name to a list. This means we have to keep the temporary file names around until all the data is processed.

Here is the code to read the input spreadsheet, followed by utility routines to assist in processing the DOM tree.

```
#
#   Extract the content.xml file from the given
#   filename, parse it, and return a DOM object.
#
sub makeDOM
{
    my ($filename) = shift;
    my $input_zip = Archive::Zip->new( $filename );
    my $parser = new XML::DOM::Parser;
    my $doc;
    my $temp_handle;
    my $temp_filename;

    ($temp_handle, $temp_filename) = Archive::Zip->tempFile();

    $input_zip->extractMember( "content.xml", $temp_filename );

    $doc = $parser->parsefile( $temp_filename );
    unlink $temp_filename;
    return $doc;
}

#
#   $node - starting node
#   $name - name of desired child element
#   returns the node's first child with the given name
#
sub getFirstChildElement ❶
{
    my ($node, $name) = @_;
    for my $child ($node->getChildNodes)
    {
        if ($child->getNodeName eq $name)
        {
            return $child;
        }
    }
    return undef;
}

#
#   $node - starting node
#   $name - name of desired sibling element
#   returns the node's next sibling with the given name
#
sub getNextSiblingElement ❷
{
    my ($node, $name) = @_;

    while (($node = $node->getNextSibling) &&
        $node->getNodeName ne $name)
    {
        # do nothing
        ;
    }
    return $node;
}

#
#   $itemref - Reference to an array to hold the row contents
#   $rowNode - a table row
#
sub getRowContents ❸
{
    my ($itemRef, $rowNode) = @_;
    my $cell;           # a cell node
    my $value;
    my $n_repeat;
    my $i;
    my $para;   # <text:p> node

    @{$itemRef} = ();
    $cell = getFirstChildElement( $rowNode, "table:table-cell" );
    while ($cell)
    {
        $n_repeat = $cell->getAttribute(
          "table:number-columns-repeated");
        $n_repeat = 1 if (!$n_repeat);

        $value = "";
        $para = getFirstChildElement( $cell, "text:p" );
        while ($para) ❹
        {
            $value .= $para->getFirstChild->getNodeValue . " ";
            $para = getNextSiblingElement( $para, "text:p" );
        }
        chop $value;

        for ($i=0; $i < $n_repeat; $i++)
        {
            push @{$itemRef}, $value;
        }
        $cell = getNextSiblingElement( $cell, "table:table-cell" );
    }
}
```

❶ Because an XML file may have newlines and tabs between elements, the first child of an element may not necessarily be another element. That means that the DOM's `getFirstChild` method might return a text node. Hence this utility routine, which bypasses text nodes and gets the specific element node that we are interested in.

❷ Similarly, the presence of newlines means we can't use the `getNextSibling` method, but must use this utility to bypass text nodes and get to the element we are interested in.

❸ Ths routine takes a `<table:table-row>` element and creates an array with all the row's values. It expands repeated cells (where the `table:number-columns-repeated` attribute is present).

❹ A table cell can contain multiple paragraphs; we concatenate them into one long string with blanks between each paragraph.

We start the main program by parsing the input file and emitting boilerplate for the `styles.xml` file, which is devoted to setting up the page dimensions.

```
if (scalar @ARGV != 2)
{
    print "Usage: $0 inputfile outputfile\n";
    exit;
}

print "Processing $ARGV[0]\n";

$doc = makeDOM( $ARGV[0] );

$zip = Archive::Zip->new();

($main_handle, $main_filename) = Archive::Zip->tempFile();
push @temp_filename, $main_filename;

print $main_handle <<"STYLEINFO";
<office:document-styles
  xmlns:office="urn:oasis:names:tc:opendocument:xmlns:office:1.0"
  xmlns:style="urn:oasis:names:tc:opendocument:xmlns:style:1.0"
  xmlns:text="urn:oasis:names:tc:opendocument:xmlns:text:1.0"
  xmlns:table="urn:oasis:names:tc:opendocument:xmlns:table:1.0"
  xmlns:draw="urn:oasis:names:tc:opendocument:xmlns:drawing:1.0"
  xmlns:fo="urn:oasis:names:tc:opendocument:xmlns:xsl-fo-compatible:1.0"
  xmlns:xlink="http://www.w3.org/1999/xlink"
  xmlns:number="urn:oasis:names:tc:opendocument:xmlns:datastyle:1.0"
  xmlns:svg="urn:oasis:names:tc:opendocument:xmlns:svg-compatible:1.0"
  xmlns:chart="urn:oasis:names:tc:opendocument:xmlns:chart:1.0"
  xmlns:dr3d="urn:oasis:names:tc:opendocument:xmlns:dr3d:1.0"
  xmlns:math="http://www.w3.org/1998/Math/MathML"
  xmlns:script="urn:oasis:names:tc:opendocument:xmlns:script:1.0"
  xmlns:dom="http://www.w3.org/2001/xml-events"
  office:version="1.0">

    <office:automatic-styles>
        <style:page-layout style:name="pm1">
            <style:page-layout-properties
              fo:page-width="21.59cm" fo:page-height="27.94cm"
              style:num-format="1" style:print-orientation="portrait"
              fo:margin-top="1.27cm" fo:margin-bottom="1.27cm"
              fo:margin-left="1.27cm" fo:margin-right="1.27cm"
              style:writing-mode="lr-tb"
              style:footnote-max-height="0cm">
                <style:columns fo:column-count="0"
                  fo:column-gap="0cm"/>
            </style:page-layout-properties>
            <style:header-style/>
            <style:footer-style/>
        </style:page-layout>
    </office:automatic-styles>
    <office:master-styles>
        <style:master-page style:name="Standard"
           style:page-layout-name="pm1"/>
    </office:master-styles>
</office:document-styles>
STYLEINFO

close $main_handle;
$zip->addFile( $main_filename, "styles.xml" );
```

The next step is to start creating the manifest file. This code is the boilerplate for the main directory files; as we create the charts, we will append elements to the manifest file.

```
#
#   Create manifest file and its boilerplate
#

($manifest_handle, $manifest_filename) = Archive::Zip->tempFile();
push @temp_filename, $manifest_filename;

print $manifest_handle <<"MANIFEST_HEADER";
<!DOCTYPE manifest:manifest
    PUBLIC "-//OpenOffice.org//DTD Manifest 1.0//EN" "Manifest.dtd">
<manifest:manifest
  xmlns:manifest="urn:oasis:names:tc:opendocument:xmlns:manifest:1.0">
   <manifest:file-entry
     manifest:media-type="application/vnd.oasis.opendocument.text"
     manifest:full-path="/"/>
   <manifest:file-entry
     manifest:media-type="text/xml" manifest:full-path="content.xml"/>
    <manifest:file-entry
      manifest:media-type="text/xml" manifest:full-path="styles.xml"/>
MANIFEST_HEADER
```

Warning

Because we are not creating a `settings.xml` file, OpenOffice.org will think your document has not been saved when you first load it.

And now, the main event: the `content.xml` file. First, the boilerplate for the styles that we will need for the text and the chart itself:

```
#
#   Create the main content.xml file and its
#   header information
#
($main_handle, $main_filename) = Archive::Zip->tempFile();
push @temp_filename, $main_filename;

print $main_handle  <<"CONTENT_HEADER";
<?xml version="1.0" encoding="UTF-8"?>
<office:document-content
  xmlns:office="urn:oasis:names:tc:opendocument:xmlns:office:1.0"
  xmlns:style="urn:oasis:names:tc:opendocument:xmlns:style:1.0"
  xmlns:text="urn:oasis:names:tc:opendocument:xmlns:text:1.0"
  xmlns:table="urn:oasis:names:tc:opendocument:xmlns:table:1.0"
  xmlns:draw="urn:oasis:names:tc:opendocument:xmlns:drawing:1.0"
  xmlns:fo="urn:oasis:names:tc:opendocument:xmlns:xsl-fo-compatible:1.0"
```

```
xmlns:xlink="http://www.w3.org/1999/xlink"
xmlns:number="urn:oasis:names:tc:opendocument:xmlns:datastyle:1.0"
xmlns:svg="urn:oasis:names:tc:opendocument:xmlns:svg-compatible:1.0"
xmlns:chart="urn:oasis:names:tc:opendocument:xmlns:chart:1.0"
xmlns:script="urn:oasis:names:tc:opendocument:xmlns:script:1.0"
office:version="1.0">
  <office:scripts/>

  <office:font-face-decls>
      <style:font-face style:name="Bitstream Charter"
          svg:font-family="&apos;Bitstream Charter&apos;"
          style:font-pitch="variable"/>
  </office:font-face-decls>

  <office:automatic-styles>
      <!-- style for question title -->
      <style:style style:name="hdr1" style:family="paragraph">
          <style:text-properties
          style:font-name="Bitstream Charter"
          style:font-family-generic="roman"
          style:font-pitch="variable"
          fo:font-size="14pt"
          fo:font-style="italic"/>
      </style:style>

      <!-- style for text summary of results -->
      <style:style style:name="info" style:family="paragraph">
          <style:paragraph-properties>
              <style:tab-stops> ❶
                  <style:tab-stop style:position="3.5cm"
                      style:type="right"/>
                  <style:tab-stop style:position="5cm"
                      style:type="char"
                      style:char="."/>
              </style:tab-stops>
          </style:paragraph-properties>
          <style:text-properties
              style:font-name="Bitstream Charter"
              style:font-family-generic="roman"
              style:font-pitch="variable"
              fo:font-size="10pt"/>
      </style:style>

      <!-- style to force a move to column two -->
      <style:style style:name="colBreak" style:family="paragraph">
          <style:paragraph-properties fo:break-before="column"/>
      </style:style>

      <!-- set column widths --> ❷
      <style:style style:name="Sect1" style:family="section">
          <style:section-properties
              text:dont-balance-text-columns="true">
              <style:columns fo:column-count="2">
                  <style:column style:rel-width="5669*"
                      fo:margin-left="0cm" fo:margin-right="0cm"/>
                  <style:column style:rel-width="5131*"
                      fo:margin-left="0cm" fo:margin-right="0cm"/>
              </style:columns>
          </style:section-properties>
      </style:style>
```

```
        <!-- style for chart frame -->
        <style:style style:name="fr1" style:family="graphic">
            <style:graphic-properties style:wrap="run-through"
                style:vertical-pos="middle"
                style:horizontal-pos="from-left"/>
        </style:style>
    </office:automatic-styles>

    <office:body>
        <office:text>
CONTENT_HEADER
```

❶ Rather than create a table for the summary of the results, we took the easy way out and set up tab stops to align the data properly.

❷ We have two columns with text that is not automatically distributed to both columns. Because the columns have different relative widths, we do *not* have an `fo:column-gap` attribute in the `<style:columns>` element.

That finishes the static portion of the content file. We now grab all the rows. Then, for each row in the table

- Find the total number of responses.
- Create a new section with the question text as the header.
- For each cell in the row, output the legend (Strongly agree, agree, etc.), the number of responses, and the percentage.
- Create a reference to the chart.
- Create a directory for the chart.
- Create the chart itself (handled in a subroutine).
- Add the path to the chart to the manifest file.

After processing all the rows, we close the remaining tags in the `content.xml` and `manifest.xml` files, and then close the files. Once all the files are created and added to the .zip file, we write the zip file and then unlink the temporary files. This finishes the main program.

```
$rows = $doc->getElementsByTagName( "table:table-row" );
getRowContents( \@legends, $rows->item(0));

$n_rows = $rows->getLength;

for ($row=1; $row<$n_rows; $row++)
{
    getRowContents( \@data, $rows->item($row));

    next if (!$data[0]);  # skip rows without a question

    $sum = 0;
    for ($col=1; $col < scalar(@data); $col++)
    {
        $sum += $data[$col];
    }
    print $main_handle qq!<text:section text:style-name="Sect1"!;
    print $main_handle qq! text:name="Section$row">!;
    print $main_handle qq!<text:h text:style-name="hdr1"►
text:outline-level="1">!;
    print $main_handle qq!$row. $data[0]</text:h>\n!;
    for ($col=1; $col < scalar(@data); $col++)
    {
        $percent = sprintf(" (%.2f%%)", 100*$data[$col]/$sum);
        print $main_handle qq!<text:p text:style-name="info">!;
        print $main_handle qq!$legends[$col]<text:tab/>$data[$col]!;
        print $main_handle qq!<text:tab/>$percent</text:p>\n!;
    }

    # now insert the reference to the graph

    print $main_handle qq!<text:p text:style-name="colBreak">!;
    print $main_handle qq!<draw:frame draw:style-name="fr1"
        draw:name="Object$row" draw:layer="layout"
        svg:width="8cm" svg:height="7cm"><draw:object
        xlink:href="./Object$row" xlink:type="simple"
        xlink:show="embed" xlink:actuate="onLoad"/></draw:frame>\n!;
    print $main_handle qq!</text:p>\n!;
    print $main_handle qq!</text:section>\n!;

    construct_chart( \@legends, \@data, $row );

    append_manifest( $row );
}

print $main_handle <<"CONTENT_FOOTER";
        </office:text>
    </office:body>
</office:document-content>
CONTENT_FOOTER

close $main_handle;

print $manifest_handle "</manifest:manifest>\n";
close $manifest_handle;
$zip->addFile( $manifest_filename, "META-INF/manifest.xml");
$zip->addFile( $main_filename, "content.xml");
$zip->writeToFileNamed( $ARGV[1] );

foreach $item (@temp_filename)
{
    unlink $item;
}
```

Let’s handle the easy subroutine first—adding the path information to the manifest file. The `append_manifest` subroutine takes one parameter: the chart number.

```
#
#   Append data to the manifest file;
#   the parameter is the chart number
#
sub append_manifest
{
    my $number = shift;
    print $manifest_handle <<"ADD_MANIFEST";
<manifest:file-entry
    manifest:media-type="application/vnd.oasis.opendocument.chart"
    manifest:full-path="Object$number/"/>
<manifest:file-entry
    manifest:media-type="text/xml"
    manifest:full-path="Object$number/content.xml"/>

ADD_MANIFEST
}
```

Finally, the subroutine to construct the chart. Again, we start with an immense amount of boilerplate, with styles for the chart title, legend, plot area, the data series, and the individual pie slices.

```
#
#   Construct the chart file, given:
#       reference to the @legends array
#       reference to the @data array
#       chart number
#
sub construct_chart
{
    my $legendref = shift;
    my $dataref = shift;
    my $chart_num = shift;

    my $cell;   # current cell number being processed

    ($chart_handle, $chart_filename) = Archive::Zip->tempFile();
    push @temp_filename, $chart_filename;
    print $chart_handle <<"CHART_HEADER";
<office:document-content
  xmlns:office="urn:oasis:names:tc:opendocument:xmlns:office:1.0"
  xmlns:style="urn:oasis:names:tc:opendocument:xmlns:style:1.0"
  xmlns:text="urn:oasis:names:tc:opendocument:xmlns:text:1.0"
  xmlns:table="urn:oasis:names:tc:opendocument:xmlns:table:1.0"
  xmlns:draw="urn:oasis:names:tc:opendocument:xmlns:drawing:1.0"
  xmlns:fo="urn:oasis:names:tc:opendocument:xmlns:xsl-fo-compatible:1.0"
  xmlns:xlink="http://www.w3.org/1999/xlink"
  xmlns:number="urn:oasis:names:tc:opendocument:xmlns:datastyle:1.0"
  xmlns:svg="urn:oasis:names:tc:opendocument:xmlns:svg-compatible:1.0"
  xmlns:chart="urn:oasis:names:tc:opendocument:xmlns:chart:1.0"
  office:version="1.0">
    <office:automatic-styles>
        <number:number-style style:name="N0">
            <number:number number:min-integer-digits="1"/>
        </number:number-style>
        <style:style style:name="main" style:family="chart">
            <style:graphic-properties draw:stroke="none"
                svg:stroke-width="0cm" svg:stroke-color="#000000"
                draw:fill-color="#ffffff"/>
        </style:style>
        <style:style style:name="title" style:family="chart">
            <style:text-properties
                style:font-family-generic="swiss"
                fo:font-size="12pt"/>
        </style:style>
        <style:style style:name="legend" style:family="chart">
            <style:text-properties
                style:font-family-generic="swiss"
                fo:font-size="8pt"/>
        </style:style>
        <style:style style:name="plot" style:family="chart">
            <style:chart-properties
                chart:lines="false"
                chart:series-source="columns"/>
        </style:style>
        <style:style style:name="axis" style:family="chart"
            style:data-style-name="N0">
            <style:chart-properties chart:display-label="true"/>
        </style:style>
        <style:style style:name="series" style:family="chart">
            <style:graphic-properties draw:fill-color="#ffffff"/>
        </style:style>

        <style:style style:name="slice1" style:family="chart">
            <style:graphic-properties draw:fill-color="#ff6060"/>
        </style:style>
        <style:style style:name="slice2" style:family="chart">
            <style:graphic-properties draw:fill-color="#ffa560"/>
        </style:style>
        <style:style style:name="slice3" style:family="chart">
            <style:graphic-properties draw:fill-color="#ffff60"/>
        </style:style>
        <style:style style:name="slice4" style:family="chart">
            <style:graphic-properties draw:fill-color="#60ff60"/>
        </style:style>
        <style:style style:name="slice5" style:family="chart">
            <style:graphic-properties draw:fill-color="#6060ff"/>
        </style:style>
        <style:style style:name="slice6" style:family="chart">
            <style:graphic-properties draw:fill-color="#606080"/>
        </style:style>
    </office:automatic-styles>
```

The “here” document continues with the static part of the `<office:body>`, setting up the chart, title, legend, plot area, and table headings. There is only one series of data per chart, and each series has six data points. The first row of the table is a dummy header row, with the letter `N` (number of responses) as its content.

```
<office:body>
  <office:chart>
    <chart:chart chart:class="chart:circle" chart:style-name="main"
      svg:width="9cm" svg:height="9cm">
    <chart:title chart:style-name="title" svg:x="1cm"
        svg:y="0.25cm">
        <text:p>${$dataref}[0]</text:p>
    </chart:title>
    <chart:legend chart:legend-position="end" svg:x="8cm" svg:y="3cm"
        chart:style-name="legend"/>

    <chart:plot-area svg:x="0.5cm" svg:y="1.5cm"
        svg:width="6cm" svg:height="6cm" chart:style-name="plot">
        <chart:axis chart:dimension="y" chart:style-name="axis"
            chart:name="primary-y">
            <chart:grid chart:class="major"/>
        </chart:axis>
        <chart:series chart:style-name="series"
            chart:values-cell-range-address="local-table.B2:.B7"
            chart:label-cell-address="local-table.B1">
            <chart:data-point chart:style-name="slice1"/>
            <chart:data-point chart:style-name="slice2"/>
            <chart:data-point chart:style-name="slice3"/>
            <chart:data-point chart:style-name="slice4"/>
            <chart:data-point chart:style-name="slice5"/>
            <chart:data-point chart:style-name="slice6"/>
        </chart:series>
    </chart:plot-area>
    <table:table table:name="local-table">
        <table:table-header-columns>
            <table:table-column/>
        </table:table-header-columns>
        <table:table-columns>
            <table:table-column table:number-columns-repeated="2"/>
        </table:table-columns>

        <table:table-header-rows>
            <table:table-row>
                <table:table-cell><text:p/></table:table-cell>
                <table:table-cell office:value-type="string">
                    <text:p>N</text:p>
                </table:table-cell>
            </table:table-row>
        </table:table-header-rows>
        <table:table-rows>
CHART_HEADER
```

Now we create the dynamic portion of the table contents; each category (Strongly Agree/Agree/etc.) is in the first column, and the number of responses in the second column. The subroutine finishes by closing off all the open tags.

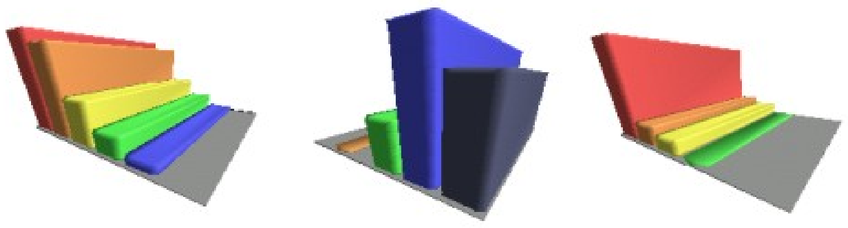
```
    for ($cell=1; $cell < scalar(@{$dataref}); $cell++)
    {
        print $chart_handle qq!<table:table-row>\n!;
        print $chart_handle qq!<table:table-cell ►
office:value-type="string">!;
        print $chart_handle qq!<text:p>!, ${$legendref}[$cell],
           qq!</text:p>!;
        print $chart_handle qq!</table:table-cell>!;
        print $chart_handle
            qq!<table:table-cell office:value-type="float" !;
        print $chart_handle qq!office:value="!,
            ${$dataref}[$cell], qq!">!;
        print $chart_handle qq!<text:p>!,
            ${$dataref}[$cell], qq!</text:p>!;
        print $chart_handle
            qq!</table:table-cell></table:table-row>\n!;
    }
    print $chart_handle <<"CHART_FOOTER";
</table:table-rows>
</table:table>
</chart:chart>
</office:chart>
</office:body>
</office:document-content>
```

```
CHART_FOOTER
    close $chart_handle;
    $zip->addFile( $chart_filename, "Object$row/content.xml" );
}
```

Three-D Charts

To make a three-dimensional chart, you must add the `chart:three-dimensional` attribute to the style that controls the `<chart:plot-area>`, and you must give it a value of `true`. If you want extra depth on the chart, you may set the `chart:deep` attribute to `true` as well. In a perfect world, that would be all that you would need to do. Unfortunately, if you leave it at that, your three-d bar charts will come out looking like Figure 8.7, "Insufficient Three-Dimensional Information", which is not what you want.[14]

Figure 8.7. Insufficient Three-Dimensional Information



In order to get a reasonable-looking chart, you must add the following attributes to your `<chart:plot-area>` element. You can get by with just the first of these, `<dr3d:distance>`, but the results will still be significantly distorted.

- `dr3d:distance`, the distance from the camera to the object.
- `dr3d:focal-length`, the length of focus of the virtual camera.
- `dr3d:projection`, which may be either `parallel` or `perspective`.

You may also add any of the attributes that you would add to a `<dr3d:scene>` element, as described in the section called "The dr3d:scene element". If you want to set the lighting, add `<dr3d:light>` elements as children of the `<chart:plot-area>` element. This element is described in the section called "Lighting".

[14] As modern art, this is actually quite nice. The results for a pie chart are quite disturbing.

Chapter 9. Filters in OpenOffice.org

To this point, we have been building stand-alone applications to transform external files, in XML format or just plain text, to OpenDocument format. OpenOffice.org allows you to integrate an XSLT transformation into the application as a *filter*.

XSLT-based filters work by associating an XML file type, which we will call the “foreign” file, XSLT transformation files for import and/or export, and an OpenOffice.org template file. XML elements in the foreign file are associated with styles in the template file. The import transformation will take the foreign file’s content and insert it into the template, assigning styles as appropriate. The export transformation will read the OpenOffice.org document, and, using the style information, create a foreign file.

The remainder of this chapter will be a case study that shows how to construct and install XSLT-based filters.

The Foreign File Format

The XML that we will import is a database of amateur wrestling clubs in California (yes, this is an actual database; the phone numbers and emails have been changed.) The state is divided into several areas or *associations*; for example, SCVWA—the Santa Clara Valley Wrestling Association. Each association consists of a series of *clubs*. Example 9.1, “Sample Club Database” shows an abbreviated file. A club can have multiple email addresses, and the `<info>` element is optional. The only element that isn’t self-explanatory is the `<age-groups>` element. Its `type` attribute tells which age groups the club serves: Kids, Cadets, Juniors, Open (competitors out of high school), and Women. The `<info>` element may contain hypertext link to a club’s website, represented by the HTML `<a>` element, which has been borrowed into this custom language without a namespace.

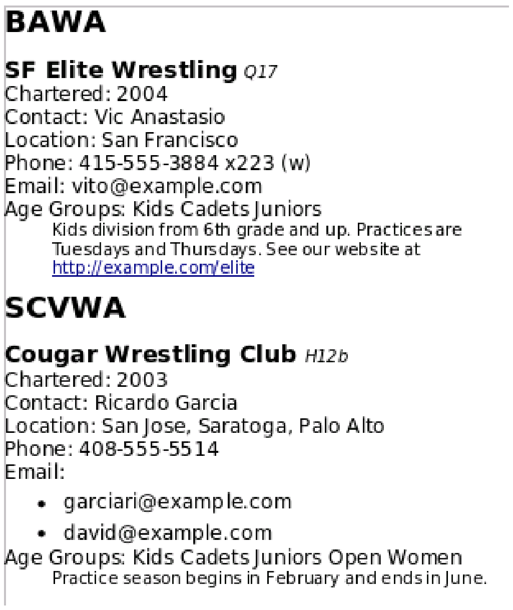
Example 9.1. Sample Club Database

```
<club-database>
<association id="BAWA">
<club id="Q17" charter="2004">
    <name>SF Elite Wrestling</name>
    <contact>Vic Anastasio</contact>
    <location>San Francisco</location>
    <phone>415-555-3884 x223 (w)</phone>
    <email>vito@example.com</email>
    <age-groups type="KCJ"/>
    <info>
        Kids division from 6th grade and up.
        Practices are Tuesdays and Thursdays. See our website at
        <a►
 href="http://example.com/elite">http://example.com/elite</a>
    </info>
</club>
```

```
</association>

<association id="SCVWA">
<club id="H12b" charter="2003">
    <name>Cougar Wrestling Club</name>
    <contact>Ricardo Garcia</contact>
    <location>San Jose, Saratoga, Palo Alto</location>
    <phone>408-555-5514</phone>
    <email>garciari@example.com</email>
    <email>david@example.com</email>
    <age-groups type="KCJOW"/>
    <info>
        Practice season begins in February and ends in June.
    </info>
</club>
</association>
</club-database>
```

Figure 9.1, "Imported Club Database" shows the OpenOffice.org Writer file that we want as a result.

Figure 9.1. Imported Club Database


BAWA

SF Elite Wrestling *Q17*
Chartered: 2004
Contact: Vic Anastasio
Location: San Francisco
Phone: 415-555-3884 x223 (w)
Email: vito@example.com
Age Groups: Kids Cadets Juniors
Kids division from 6th grade and up. Practices are Tuesdays and Thursdays. See our website at http://example.com/elite

SCVWA

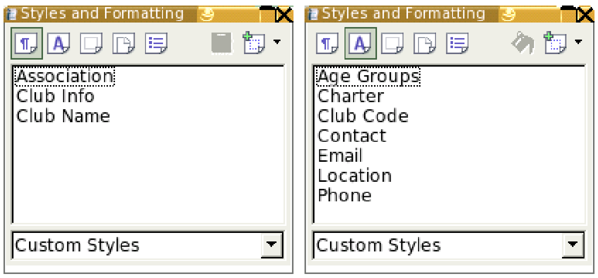
Cougar Wrestling Club *H12b*
Chartered: 2003
Contact: Ricardo Garcia
Location: San Jose, Saratoga, Palo Alto
Phone: 408-555-5514
Email:
- garciari@example.com
- david@example.com

Age Groups: Kids Cadets Juniors Open Women
Practice season begins in February and ends in June.

Building the Import Filter

We will now create the template file in OpenOffice.org. This is just a skeleton document with styles that will be associated with XML elements. Figure 9.2, “Styles in Writer Template” shows the names of the paragraph and character styles in the template. [This is file `clublist_template.ott` in directory `ch09` in the downloadable example files.]

Figure 9.2. Styles in Writer Template



Association
Club Info
Club Name

Age Groups
Charter
Club Code
Contact
Email
Location
Phone

That having been done, we create the stylesheet, shown in Example 9.2, “Stylesheet for Transforming Club List to Writer Document”. The template doesn’t have to include any `<style:style>` elements; those have been taken care of in the template. [This is file `club_to_writer.xsl` in directory `ch09` in the downloadable example files.]

Example 9.2. Stylesheet for Transforming Club List to Writer Document

```
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:office="urn:oasis:names:tc:opendocument:xmlns:office:1.0"
  xmlns:style="urn:oasis:names:tc:opendocument:xmlns:style:1.0"
  xmlns:text="urn:oasis:names:tc:opendocument:xmlns:text:1.0"
  xmlns:table="urn:oasis:names:tc:opendocument:xmlns:table:1.0"
  xmlns:draw="urn:oasis:names:tc:opendocument:xmlns:drawing:1.0"
  xmlns:fo="urn:oasis:names:tc:opendocument:xmlns:xsl-fo-compatible:1.0"
  xmlns:xlink="http://www.w3.org/1999/xlink"
  xmlns:dc="http://purl.org/dc/elements/1.1/"
  xmlns:meta="urn:oasis:names:tc:opendocument:xmlns:meta:1.0"
  xmlns:number="urn:oasis:names:tc:opendocument:xmlns:datastyle:1.0"
  xmlns:svg="urn:oasis:names:tc:opendocument:xmlns:svg-compatible:1.0"
  xmlns:chart="urn:oasis:names:tc:opendocument:xmlns:chart:1.0"
  xmlns:math="http://www.w3.org/1998/Math/MathML"
  xmlns:form="urn:oasis:names:tc:opendocument:xmlns:form:1.0"
  xmlns:script="urn:oasis:names:tc:opendocument:xmlns:script:1.0"
  office:version="1.0">

<xsl:template match="/">
    <office:document>
        <office:body>
            <xsl:apply-templates select="club-database/association"/>
        </office:body>
    </office:document>
```

```
</xsl:template>

<xsl:template match="association">
    <text:h text:outline-level="1" text:style-name="Association"> ❶
        <xsl:value-of select="@id"/>
    </text:h>
    <xsl:apply-templates select="club"/>
</xsl:template>


<xsl:template match="club">
    <text:h text:level="2" text:style-name="Club Name">
        <xsl:value-of select="name" />
        <xsl:text> </xsl:text>
        <text:span text:style-name="Club Code"><xsl:value-of ❷
            select="@id" /></text:span>
    </text:h>
    <text:p text:style-name="Default">
        <xsl:text>Chartered: </xsl:text>
        <text:span text:style-name="Charter">
            <xsl:value-of select="@charter"/>
        </text:span>
    </text:p>
    <text:p text:style-name="Default">
        <xsl:text>Contact: </xsl:text>
        <text:span text:style-name="Contact">
            <xsl:value-of select="contact"/>
        </text:span>
    </text:p>
    <text:p text:style-name="Default">
        <xsl:text>Location: </xsl:text>
        <text:span text:style-name="Location">
            <xsl:value-of select="location"/>
        </text:span>
    </text:p>
    <text:p text:style-name="Default">
        <xsl:text>Phone: </xsl:text>
        <text:span text:style-name="Phone">
            <xsl:value-of select="phone"/>
        </text:span>
    </text:p>

    <xsl:choose>
        <xsl:when test="count(email) = 1"> ❸
            <text:p text:style-name="Default">
                <xsl:text>Email: </xsl:text>
                <text:span text:style-name="Email">
                    <xsl:value-of select="email"/>
                </text:span>
            </text:p>
        </xsl:when>
        <xsl:when test="count(email) &gt; 1">
            <text:p text:style-name="Default">
                <text:span>Email:</text:span>
            </text:p>
            <text:list text:style-name="UnorderedList">
                <xsl:for-each select="email">
                    <text:list-item>
                        <text:p text:style-name="Default">
                            <text:span text:style-name="Email">
```

```
                                <xsl:value-of select="."/>
                            </text:span>
                        </text:p>
                    </text:list-item>
                </xsl:for-each>
            </text:list>
        </xsl:when>
    </xsl:choose>

    <xsl:apply-templates select="age-groups"/>

    <xsl:apply-templates select="info"/>
</xsl:template>

<xsl:template match="age-groups">
    <text:p text:style-name="Default">
        <xsl:text>Age Groups: </xsl:text>
        <text:span text:style-name="Age Groups">
            <xsl:if test="contains(@type,'K')"> ❹
                <xsl:text>Kids </xsl:text>
            </xsl:if>
            <xsl:if test="contains(@type,'C')">
                <xsl:text>Cadets </xsl:text>
            </xsl:if>
            <xsl:if test="contains(@type,'J')">
                <xsl:text>Juniors </xsl:text>
            </xsl:if>
            <xsl:if test="contains(@type,'O')">
                <xsl:text>Open </xsl:text>
            </xsl:if>
            <xsl:if test="contains(@type,'W')">
                <xsl:text>Women </xsl:text>
            </xsl:if>
        </text:span>
    </text:p>
</xsl:template>

<xsl:template match="info">
    <text:p text:style-name="Club Info"> ❺
        <xsl:if test="normalize-space(.) != ''">
            <xsl:apply-templates/>
        </xsl:if>
    </text:p>
</xsl:template>

<xsl:template match="a"> ❻
    <text:a xlink:type="simple" xlink:href="{@href}"><xsl:value-of
select="."/></text:a>
</xsl:template>

</xsl:stylesheet>
```

❶ This is the first occurrence of connecting the foreign file’s content with a custom style in the template.

❷ Notice that we attach the style only to the actual content, not to the entire paragraph. This means we don’t have to parse the paragraph content upon export.

❸ If there’s only one email address, it is placed on the same line as the label; otherwise, the transformation creates an unordered list of all the email addresses.

❹ Go through the age group symbols one at a time. Note that we *will* have to parse this in the export transformation.

❺ Even if there’s nothing in the `<info>` element, we want an empty paragraph for the spacing.

❻ This is how you add a hypertext link to an OpenOffice.org Writer document; it also borrows the `<a>` element from HTML, but does it the right way—with a namespace.

Building the Export Filter

Creating the export filter is a much more difficult task. When we imported a file, a hierarchical structure like this …

```
<association>
    <club>
        <name />
        <contact />
    </club>
    <club>
        <name />
        <contact />
    </club>
</association>
<association>
    <!-- etc -->
</association>
```

… was “flattened” into a structure like this:

```
<text:h text:style-name="Association"/>
<text:h text:style-name="Club Name"/>
<text:p>Contact: <text:span text:style-name="Contact"/></text:p>
<text:h text:style-name="Club Name"/>
<text:p>Contact: <text:span text:style-name="Contact"/></text:p>
<text:h text:style-name="Association"/>
<!-- etc -->
```

The export filter will have to take this flattened structure and re-create the nesting. The algorithm for this is not particularly difficult:

For each `<text:h>` element with a `text:style-name` of `Association`:

1. Open an `<association>` element.
2. While the next `<text:h>` element has a `text:style-name` of `Club Name`, construct a `<club>` element (see following pseudocode).
3. Close the `<association>` element.

To construct a `<club>` element:

1. Create an opening `<club>` element.
2. While the next sibling of this element is a `<text:p>` element:
    a. If there is a child `<text:span>` element, create an appropriate child element based on the span's `text:style-name`.
    b. Otherwise, if there is a neighboring `<text:list>`, then you have a list of emails.[15] Extract the email addresses and create the appropriate `<email>` elements in the target document.
    c. Otherwise, if this is a club info paragraph, inset an `<info>` element.
3. You have encountered a `<text:h>` element or the end of the file. Close the `<club>` element.

This is not exactly rocket surgery, but the job is complicated by the fact that XSLT almost exclusively uses recursion, not iteration.[16] This makes the transformation ugly, so we will present it in parts. [This is file `writer_to_club.xsl` in directory `ch09` in the downloadable example files.]

The first part shows the opening `<xsl:stylesheet>` element, showing the namespaces that could be used in the OpenOffice.org document. The transformation won't work without these declarations, but we do not want to see the namespaces in the resulting output file. Thus, we use the `exclude-result-prefixes` attribute to eliminate namespace delcarations from our ouput.

```
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:fo="urn:oasis:names:tc:opendocument:xmlns:xsl-fo-compatible:1.0"
  xmlns:office="urn:oasis:names:tc:opendocument:xmlns:office:1.0"
  xmlns:style="urn:oasis:names:tc:opendocument:xmlns:style:1.0"
  xmlns:text="urn:oasis:names:tc:opendocument:xmlns:text:1.0"
  xmlns:table="urn:oasis:names:tc:opendocument:xmlns:table:1.0"
  xmlns:draw="urn:oasis:names:tc:opendocument:xmlns:drawing:1.0"
  xmlns:xlink="http://www.w3.org/1999/xlink"
  xmlns:number="urn:oasis:names:tc:opendocument:xmlns:datastyle:1.0"
  xmlns:form="urn:oasis:names:tc:opendocument:xmlns:form:1.0"
  xmlns:script="urn:oasis:names:tc:opendocument:xmlns:script:1.0"
  xmlns:svg="urn:oasis:names:tc:opendocument:xmlns:svg-compatible:1.0"
  exclude-result-prefixes="text xsl fo office style table draw xlink
form script config number svg">
<xsl:output method="xml" indent="yes"/>
```

[15] This is where our cleverness of representing multiple emails as a list comes back to haunt us.

[16] When your only tool is a hammer, everything looks like a nail.

Almost the only place we can use XSLT's natural processing style is to grab all the `<text:h>` elements for the associations. Processing an association creates the `<association>` element with its ID, and then starts the process of making entries for the constituent clubs. Implicit in this code is the presumption that there is at least one club in an association.

> **Note**
> When you are exporting a document, its XML representation is a "unified document," with the contents of *all* the files (`meta.xml`, `styles.xml`, `content.xml`, etc.) all enclosed in an `<office:document>` element, not the `<office:document-content>` that we have been using in previous chapters. If you want to see what such a file looks like, install file `unified_document.xsl` in directory `ch09` from the downloadable example files.

```
<xsl:template match="/">
    <xsl:apply-templates select="office:document/office:body/
        office:text/text:h[@text:style-name='Association']"/>
</xsl:template>

<xsl:template match="text:h[@text:style-name='Association']">
    <association id="{.}">
        <xsl:call-template name="make-club">
            <xsl:with-param name="clubNode"
             select="following-sibling::text:h[1]"/>
        </xsl:call-template>
    </association>
</xsl:template>
```

We can now make the club(s) in the association.

```
<xsl:template name="make-club">
    <xsl:param name="clubNode"/>
    <xsl:if test="$clubNode/@text:style-name = 'Club_20_Name'"> ❶
         <club>
            <xsl:attribute name="id">
              <xsl:value-of
                  select="$clubNode/text:span[@text:
                  style-name='Club_20_Code']"/>
            </xsl:attribute>
            <name><xsl:value-of select="$clubNode"/></name>
            <xsl:call-template name="make-content">
                <xsl:with-param name="contentNode"
                    select="$clubNode/following-sibling::*[1]"/> ❷
            </xsl:call-template>

        </club>
        <xsl:if test="$clubNode/following-sibling::text:h[1]"> ❸
            <xsl:call-template name="make-club">
                <xsl:with-param name="clubNode"
                    select="$clubNode/following-sibling::text:h[1]"/>
            </xsl:call-template>
        </xsl:if>
    </xsl:if>
</xsl:template>
```

❶ The node that was passed on to the `make-club` template could be either a `<text:h>` for a club name or the next association if this was the last club. Hence, the `<xsl:if>` to make sure we have a club name.

❷ When we proceed to gather the club's content, we have to blindly pass on the first following sibling element—it could be a `<text:p>` that is part of the club, a `<text:h>` that starts a new club, or a `<text:h>` that starts a new association.

❸ After completing this club, check to see if this node has a following `<text:h>` node. If so, recursively call this template with that new node, which could be another club or the next association.

Assembling the content for a club works very much along the same lines.

```
<xsl:template name="make-content">
    <xsl:param name="contentNode"/>
    <xsl:if test="name($contentNode) = 'text:p'"> ❶
      <xsl:choose>
          <xsl:when test="$contentNode/text:span"> ❷
              <xsl:call-template name="add-item">
                  <xsl:with-param name="spanNode"
                      select="$contentNode/text:span"/>
              </xsl:call-template>
          </xsl:when>
          <xsl:when test="name($contentNode/following-sibling::*[1]) =
                'text:list'"> ❸
              <xsl:call-template name="email-list">
                  <xsl:with-param name="emailList"
                      select="$contentNode/following-sibling::
                      text:list[1]"/>
              </xsl:call-template>
          </xsl:when>
          <xsl:when test="$contentNode/@text:style-name =
              'Club_20_Info'"> ❹
              <info>
                  <xsl:apply-templates select="$contentNode"/>
              </info>
          </xsl:when>
      </xsl:choose>
      <xsl:call-template name="make-content"> ❺
          <xsl:with-param name="contentNode"
              select="$contentNode/following-sibling::*[1]"/>
      </xsl:call-template>
    </xsl:if>
</xsl:template>
```

❶ If this isn't a paragraph, then it's not part of the club content. (This stops recursion when we hit the end of the file or the next club/association.)

❷ If this paragraph has a `<text:span>` child, then it's a charter, location, contact, phone, single email, or age group specification. Hand it off to another template.

❸ If there's an unordered list following this paragraph, then it must be a club with multiple emails. Again, hand the list off to another template.

❹ Club information is just straight text with embedded links, so use `<apply-templates>` to handle the text (with the default template) and the links with a soon-to-be-described template.

❺ In any case, keep gathering content by recursively calling this template with the next node in the document.

Here's the template that adds individual elements as children of a club. The `styleAttr` variable is for convenience, to make the source easier to read. All the elements except `<age-groups>` are handled by adding the span's contents. Age groups are special, and, rather than trying to split up a list of keywords and recursively handle them, we cheat. The call to the `translate` function eliminates all lowercase letters and blanks, leaving the uppercase abbreviations for the age groups. For example, `Kids Cadets Open` is instantly reduced to `KCO`.

```
<xsl:template name="add-item">
    <xsl:param name="spanNode"/>
    <xsl:variable name="styleAttr" select="$spanNode/@text:style-name"/>

    <xsl:choose>
        <xsl:when test="$styleAttr = 'Charter'">
            <charter><xsl:value-of select="$spanNode"/></charter>
        </xsl:when>
        <xsl:when test="$styleAttr = 'Contact'">
            <contact><xsl:value-of select="$spanNode"/></contact>
        </xsl:when>
        <xsl:when test="$styleAttr = 'Phone'">
            <phone><xsl:value-of select="$spanNode"/></phone>
        </xsl:when>
        <xsl:when test="$styleAttr = 'Location'">
            <location><xsl:value-of select="$spanNode"/></location>
        </xsl:when>
        <xsl:when test="$styleAttr = 'Email'">
            <email><xsl:value-of select="$spanNode"/></email>
        </xsl:when>
        <xsl:when test="$styleAttr = 'Age_20_Groups'">
            <age-groups>
                <xsl:attribute name="type">
                    <xsl:value-of select="translate($spanNode,
                    ' abcdefghijklmnopqrstuvwxyz', '')"/>
                </xsl:attribute>
            </age-groups>
        </xsl:when>
    </xsl:choose>
</xsl:template>
```

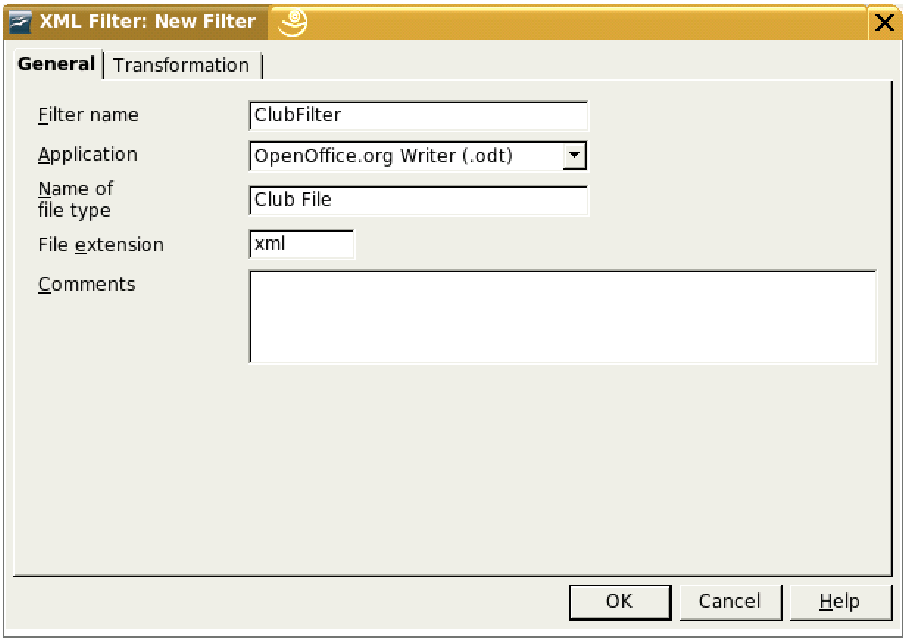
Rounding out the XSLT stylesheet are the templates that handle a list of email addresses within a `<text:unordered-list>` and the `<text:a>` element inside the club information.

```
<xsl:template name="email-list">
    <xsl:param name="emailList"/>
    <xsl:for-each
      select="$emailList/descendant::text:span[
        @text:style-name='Email']">
        <email><xsl:value-of select="."/></email>
    </xsl:for-each>
</xsl:template>

<xsl:template match="text:a">
<a href="{@xlink:href}"><xsl:apply-templates/></a>
</xsl:template>

</xsl:stylesheet>
```

Installing a Filter

To install an XSLT-based filter, you choose the “XML Filter Settings” option from the “Tools” menu. Figure 9.3, “General Filter Information” shows the entries for the club file filter.

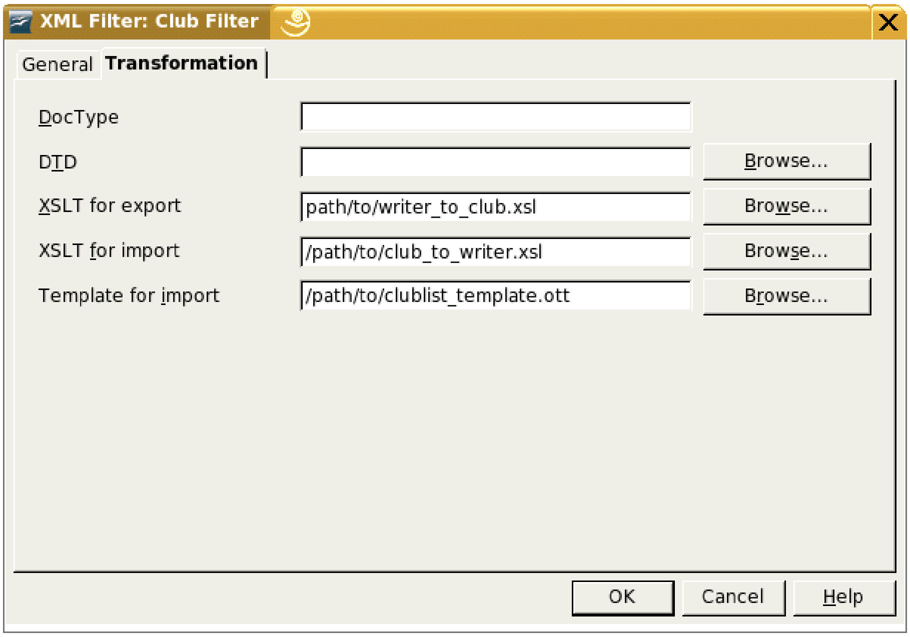
Figure 9.3. General Filter Information



Filter name: ClubFilter
Application: OpenOffice.org Writer (.odt)
Name of file type: Club File
File extension: xml
Comments:

The rest of the information about the filter is placed in the dialog for the Transformation tab, as shown in Figure 9.4, “Filter Transformation Information”.

Figure 9.4. Filter Transformation Information



DocType:
DTD:
XSLT for export: path/to/writer_to_club.xsl
XSLT for import: /path/to/club_to_writer.xsl
Template for import: /path/to/clublist_template.ott

Warning

When you first create a filter, you may specify any path name to the template. OpenOffice.org will move the template to the `*path/to/userdir*/user/template/*template name*` folder for you. (The path is the path specified in the user's directory for templates in OpenOffice.org path options dialog box.) If you update your template, you have to re-enter its original path, and OpenOffice.org will update the template in its template directory.

That's all there is to it; your new filter is ready for use. If you wish, you may also package the XSLT transformations and the template into a `.jar` file so that other users may install all the files in one swell foop by clicking the "Open package..." button in the main XML Filter Settings dialog. This will put the template in the *path/to/userdir*`/user/template/`*`template name`* directory and the XSLT file(s) into the *path/to/userdir*`/user/xslt/`*`template name`* directory.

Appendix A. The XML You Need for OpenDocument

The purpose of this appendix is to introduce you to XML. A knowledge of XML is essential if you wish to manipulate OpenDocument files directly, since XML is the basis of the OpenDocument format.

If you're already acquainted with XML, you don't need to read this appendix. If not, read on. The general overview of XML given in this appendix should be more than sufficient to enable you to work with OpenDocument documents. For further information about XML, the O'Reilly books *Learning XML* by Erik T. Ray and *XML in a Nutshell* by Elliotte Rusty Harold and W. Scott Means are invaluable guides, as is the weekly online magazine `XML.com`.

Note that this appendix makes frequent reference to the formal XML 1.0 specification, which can be used for further investigation of topics that fall outside the scope of this book. Readers are also directed to the "Annotated XML Specification,"written by Tim Bray and published online at `http://XML.com/`, which provides illuminating explanation of the XML 1.0 specification; and to "What is XML?" by Norm Walsh, also published on `XML.com`.

What is XML?

XML, the Extensible Markup Language, is an Internet-friendly format for data and documents, invented by the World Wide Web Consortium (W3C). The "Markup" denotes a way of expressing the structure of a document within the document itself. XML has its roots in a markup language called SGML (Standard Generalized Markup Language), which is used in publishing and shares this heritage with HTML. XML was created to do for machine-readable documents on the Web what HTML did for human-readable documents - that is, provide a commonly agreed-upon syntax so that processing the underlying format becomes a commodity and documents are made accessible to all users.

Unlike HTML, though, XML comes with very little predefined. HTML developers are accustomed both to the notion of using angle brackets < > for denoting elements (that is, *syntax)*, and also to the set of element names themselves (such as head, body, etc.). XML only shares the former feature (i.e., the notion of using angle brackets for denoting elements). Unlike HTML, XML has no predefined elements, but is merely a set of rules that lets you write other languages like HTML.[17]

[17] To clarify XML's relationship with SGML: XML is an *SGML subset*. By contrast, HTML is an *SGML application*. OpenDocument uses XML to express its operations and thus is an *XML application*.

Because XML defines so little, it is easy for everyone to agree to use the XML syntax, and then to build applications on top of it. It's like agreeing to use a particular alphabet and set of punctuation symbols, but not saying which language to use. However, if you're coming to XML from an HTML background, then prepare yourself for the shock of having to choose what to call your tags!

Knowing that XML's roots lie with SGML should help you understand some of XML's features and design decisions. Note that, although SGML is essentially a document-centric technology, XML's functionality also extends to data-centric applications, including OpenDocument. Commonly, data-centric applications do not need all the flexibility and expressiveness that XML provides and limit themselves to employing only a subset of XML's functionality.

Anatomy of an XML Document

The best way to explain how an XML document is composed is to present one. The following example shows an XML document you might use to describe two authors:

```
<?xml version="1.0" encoding="us-ascii"?>
<authors>
    <person id="lear">
        <name>Edward Lear</name>
        <nationality>British</nationality>
    </person>
    <person id="asimov">
        <name>Isaac Asimov</name>
        <nationality>American</nationality>
    </person>
    <person id="mysteryperson"/>
</authors>
```

The first line of the document is known as the *XML declaration*. This tells a processing application which version of XML you are using—the version indicator is mandatory[18] —and which character encoding you have used for the document. In the previous example, the document is encoded in ASCII. (The significance of character encoding is covered later in this chapter.) If the XML declaration is omitted, a processor will make certain assumptions about your document. In particular, it will expect it to be encoded in UTF-8, an encoding of the Unicode character set. However, it is best to use the XML declaration wherever possible, both to avoid confusion over the character encoding and to indicate to processors which version of XML you're using.

[18] For reasons that will be clearer later, constructs such as `version` in the XML declaration are known as *pseudoattributes*.

Elements and Attributes

The second line of the example begins an *element*, which has been named "authors." The contents of that element include everything between the right angle bracket (>) in `<authors>` and the left angle bracket (<) in `</authors>`. The actual syntactic constructs `<authors>` and `</authors>` are often referred to as the element *start tag* and *end tag*, respectively. Do not confuse tags with elements! Note that elements may include other elements, as well as text. An XML document must contain exactly one *root element*, which contains all other content within the document. The name of the root element defines the type of the XML document.

Elements that contain both text and other elements simultaneously are classified as *mixed content*. Many OpenDocument elements contain mixed content.

The sample “authors” document uses elements named `person` to describe the authors themselves. Each person element has an *attribute* named `id`. Unlike elements, attributes can only contain textual content. Their values must be surrounded by quotes. Either single quotes (`'`) or double quotes (`"`) may be used, as long as you use the same kind of closing quote as the opening one.

Within XML documents, attributes are frequently used for *metadata* (i.e., “data about data”)–describing properties of the element’s contents. This is the case in our example, where `id` contains a unique identifier for the person being described.

As far as XML is concerned, it does not matter in which order attributes are presented in the element start tag. For example, these two elements contain exactly the same information as far as an XML 1.0 conformant processing application is concerned:

```
<animal name="dog" legs="4"/>
<animal legs="4" name="dog"/>
```

On the other hand, the information presented to an application by an XML processor on reading the following two lines will be different for each animal element because the ordering of elements is significant:

```
<animal><name>dog</name><legs>4</legs></animal>
<animal><legs>4</legs><name>dog</name></animal>
```

XML treats a set of attributes like a bunch of stuff in a bag–there is no implicit ordering–while elements are treated like items on a list, where ordering matters.

New XML developers frequently ask when it is best to use attributes to represent information and when it is best to use elements. As you can see from the “authors” example, if order is important to you, then elements are a good choice. In general, there is no hard-and-fast “best practice” for choosing whether to use attributes or elements.

The final author described in our document has no information available. All we know about this person is his or her ID, `mysteryperson`. The document uses the

XML shortcut syntax for an empty element. The following is a reasonable alternative:

```
<person id="mysteryperson"></person>
```

Name Syntax

XML 1.0 has certain rules about element and attribute names. In particular:

- Names are case-sensitive: e.g., <person/> is not the same as <Person/>.
- Names beginning with “xml” (in any permutation of uppercase or lowercase) are reserved for use by XML 1.0 and its companion specifications.
- A name must start with a letter or an underscore, not a digit, and may continue with any letter, digit, underscore, or period.[19]

A precise description of names can be found in Section 2.3 of the XML 1.0 specification, at the URL `http://www.w3.org/TR/REC-xml#sec-common-syn`.

Well-Formed

An XML document that conforms to the rules of XML syntax is known as *well-formed*. At its most basic level, well-formedness means that elements should be properly matched, and all opened elements should be closed. A formal definition of well-formedness can be found in Section 2.1 of the XML 1.0 specification, at the URL `http://www.w3.org/TR/REC-xml#sec-well-formed`. Table A.1, “ Examples of poorly formed XML documents ” shows some XML documents that are not well-formed.

Table A.1. Examples of poorly formed XML documents

Document	Reason why it’s not well-formed

`<foo>`
`  <bar>`
`  </foo>`
`</bar>`
	The elements are not properly nested because `foo` is closed while inside its child element bar.

`<foo>`
`  <bar>`
`</foo>`
	The `bar` element was not closed before its parent, `foo`, was closed.

`<foo baz>`
`</foo>`
	The `baz` attribute has no value. While this is permissible in HTML (e.g., `<table border>`), it is forbidden in XML.

`<foo baz=23>`
`</foo>`
	The `baz` attribute value, `23`, has no surrounding quotes. Unlike HTML, all attribute values must be quoted in XML.

[19] Actually, a name may also contain a colon, but the colon is used to delimit a *namespace prefix* and is not available for arbitrary use. (the section called “XML Namespaces” discusses namespaces in more detail.)

Comments

As in HTML, it is possible to include *comments* within XML documents. XML comments are intended to be read only by people. With HTML, developers have occasionally employed comments to add application-specific functionality–for example, the server-side include functionality of most web servers uses instructions embedded in HTML comments. XML provides other means of indicating application processing instructions,[20] so comments should not be used for any purpose other than those for which they were intended.

The start of a comment is indicated with `<!--`, and the end of the comment with `-->`. Any sequence of characters, aside from the string `--`, may appear within a comment.

Comments tend to be used more in XML documents intended for human consumption than those intended for machine consumption. Since OpenDocument files are almost always intended for machine consumption, they will contain few if any comments.

Entity References

Another feature of XML with which is occasionally useful when creating XML documents is the mechanism for *escaping* characters.

Because some characters have special significance in XML, there needs to be a way to represent them. For example, in some cases the < symbol might really be intended to mean “less than” rather than to signal the start of an element name. Clearly, just inserting the character without any escaping mechanism would result in a poorly formed document because a processing application would assume you were commencing another element. Another instance of this problem is that of needing to include both double quotes and single quotes simultaneously in an attribute’s value. Here’s an example that illustrates both these difficulties:

```
<badDoc>
  <para>
    I'd really like to use the < character
  </para>
  <note title="On the proper 'use' of the " character"/>
</badDoc>
```

XML avoids this problem by the use of the (fearsomely named) *predefined entity references*. The word *entity* in the context of XML simply means a unit of content. The term *entity reference* means just that, a symbolic way of referring to a certain unit of content. XML predefines entities for the following symbols: left angle bracket (<), right angle bracket (>), apostrophe (`'`), double quote (`"`), and ampersand (`&`).

[20] A discussion of *processing instructions (PIs)* is outside the scope of this book. For more information on PIs, see Section 2.6 of the XML 1.0 specification, at the URL `http://www.w3.org/TR/REC-xml#sec-pi`.

An entity reference is introduced with an ampersand (`&`), which is followed by a name (using the word "name" in its formal sense, as defined by the XML 1.0 specification), and terminated with a semicolon (`;`). Table A.2, "Predefined entity references in XML 1.0" shows how the five predefined entities can be used within an XML document.

Table A.2. Predefined entity references in XML 1.0

Literal character	Entity reference
`<`	`&lt;`
`>`	`&gt;`
`'`	`&apos;`
`"`	`&quot;`
`&`	`&amp;`

Here's our problematic document revised to use entity references:

```
<badDoc>
  <para>
    I'd really like to use the &lt; character
  </para>
  <note title="On the proper &apos;use&apos; of the &quot;character"/>
</badDoc>
```

Being able to use the predefined entities is all you need for OpenDocument; in general, entities are provided as a convenience for human-created XML. XML 1.0 allows you to define your own entities and use entity references as "shortcuts" in your document. Section 4 of the XML 1.0 specification, available at `http://www.w3.org/TR/REC-xml#sec-physical-struct`, describes the use of entities.

Character References

You are likely to find *character references* in the context of XML documents that aren't generated by OpenDocument-compatible applications. Character references allow you to denote a character by its numeric position in Unicode character set (this position is known as its *code point*). Table A.3, " Example character references in UTF-8 " contains a few examples that illustrate the syntax.

Table A.3. Example character references in UTF-8

Actual character	Character reference
1	`&#48;`
A	`&#65;`
Ñ	`&#xD1;`
®	`&#xAE;`

Note that the code point can be expressed in decimal or, with the use of `x` as a prefix, in hexadecimal, as shown in the last example in Table A.3, “Example character references in UTF-8”.

Character Encodings

The subject of character encodings is frequently a mysterious one for developers. Most code tends to be written for one computing platform and, normally, to run within one organization. Although the Internet is changing things quickly, most of us have never had cause to think too deeply about internationalization.

XML, designed to be an Internet-friendly syntax for information exchange, has internationalization at its very core. One of the basic requirements for XML processors is that they support the Unicode standard character encoding. Unicode attempts to include the requirements of all the world’s languages within one character set. Consequently, it is very large!

Unicode Encoding Schemes

Unicode 3.0 has more than 57,700 *code points*, each of which corresponds to a character.[21] If one were to express a Unicode string by using the position of each character in the character set as its encoding (in the same way as ASCII does), expressing the whole range of characters would require 4 octets[22] for each character. Clearly, if a document is written in 100 percent American English, it will be four times larger than required - all the characters in ASCII fitting into a 7-bit representation. This places a strain both on storage space and on memory requirements for processing applications.

Fortunately, two encoding schemes for Unicode alleviate this problem: *UTF-8* and *UTF-16*. As you might guess from their names, applications can process documents in these encodings in 8- or 16-bit segments at a time. When code points are required in a document that cannot be represented by one chunk, a bit-pattern is used that indicates that the following chunk is required to calculate the desired code point. In UTF-8 this is denoted by the most significant bit of the first octet being set to 1.

This scheme means that UTF-8 is a highly efficient encoding for representing languages using Latin alphabets, such as English. All of the ASCII character set is represented natively in UTF-8–an ASCII-only document and its equivalent in UTF-8 are byte-for-byte identical. OpenDocument files are always encoded in UTF-8.

[21] You can obtain charts of all these characters online by visiting `http://www.unicode.org/charts/`.

[22] An *octet* is a string of 8 binary digits, or bits. A *byte* is commonly, but not always, considered the same thing as an octet.

This knowledge will also help you debug encoding errors. One frequent error arises because of the fact that ASCII is a proper subset of UTF-8–programmers get used to this fact and produce UTF-8 documents, but use them as if they were ASCII. Things start to go awry when the XML parser processes a document containing, for example, characters from foreign languages, such as Á . Because this character cannot be represented using only one octet in UTF-8, this produces a two-octet sequence in the output document; in a non-Unicode viewer or text editor, it looks like a couple of characters of garbage.

Other Character Encodings

Unicode, in the context of computing history, is a relatively new invention. Native operating system support for Unicode is by no means widespread. For instance, although Windows NT offers Unicode support, Windows 95 and 98 do not have it.

XML 1.0 allows a document to be encoded in any character set registered with the Internet Assigned Numbers Authority (IANA). European documents are commonly encoded in one of the *ISO Latin* character sets, such as ISO-8859-1. Japanese documents commonly use *Shift-JIS*, and Chinese documents use *GB2312* and *Big 5*.

A full list of registered character sets may be found on the Web at `http://www.iana.org/assignments/character-sets`.

XML processors are not required by the XML 1.0 specification to support any more than UTF-8 and UTF-16, but most commonly support other encodings, such as US-ASCII and ISO-8859-1.

Validity

In addition to well-formedness, XML 1.0 offers another level of verification, called *validity*. To explain why validity is important, let’s take a simple example. Imagine you invented a simple XML format for your friends’ telephone numbers:

```
<phonebook>
  <person>
    <name>Albert Smith</name>
    <number>123-456-7890</number>
  </person>
  <person>
    <name>Bertrand Jones</name>
    <number>456-123-9876</number>
  </person>
</phonebook>
```

Based on your format, you also construct a program to display and search your phone numbers. This program turns out to be so useful, you share it with your friends. However, your friends aren’t so hot on detail as you are, and try to feed your program this phone book file:

```
<phonebook>
  <person>
```

```
    <name>Melanie Green</name>
    <phone>123-456-7893</phone>
  </person>
</phonebook>
```

Note that, although this file is perfectly well-formed, it doesn't fit the format you prescribed for the phone book, and you find you need to change your program to cope with this situation. If your friends had used `number` as you did to denote the phone number, and not `phone`, there wouldn't have been a problem. However, as it is, this second file is not a *valid* phonebook document.

For validity to be a useful general concept, we need a machine-readable way of saying what a valid document is; that is, which elements and attributes must be present and in what order. XML 1.0 achieves this by introducing *document type definitions* (DTDs). For the purposes of OpenDocument, you don't need to know much about DTDs. OpenDocument uses a different system called Relax-NG to specify in great detail exactly which combinations of elements and attributes make up a valid document.

Document Type Definitions (DTDs)

The purpose of a DTD is to express the allowed elements and attributes in a certain document type and to constrain the order in which they must appear within that document type. A DTD is generally composed of one file, which contains declarations defining the *element types* and *attribute lists*. (In theory, a DTD may span more than one file; however, the mechanism for including one file inside another–*parameter entities*–is outside the scope of this book.) It is common to mistakenly conflate *element* and *element types*. The distinction is that an element is the actual instance of the structure as found in an XML document, whereas the element type is the *kind of element* that the instance is.

Putting It Together

What *is* important to you is knowing how to link a document to its defining DTD. This is done with a document type declaration `<!DOCTYPE` ...>, inserted at the beginning of the XML document, after the XML declaration in our fictitious example:

```
<?xml version="1.0" encoding="us-ascii"?>
<!DOCTYPE authors SYSTEM "http://example.com/authors.dtd">
<authors>
    <person id="lear">
        <name>Edward Lear</name>
        <nationality>British</nationality>
    </person>
    <person id="asimov">
        <name>Isaac Asimov</name>
        <nationality>American</nationality>
    </person>
    <person id="mysteryperson"/>
</authors>
```

This example assumes the DTD file has been placed on a web server at *example.com*. Note that the document type declaration specifies the root element of the document, not the DTD itself. You could use the same DTD to define “person,” “name,” or “nationality” as the root element of a valid document. Certain DTDs, such as the DocBook DTD for technical documentation,[23] use this feature to good effect, allowing you to provide the same DTD for multiple document types.

A *validating XML processor* is obliged to check the input document against its DTD. If it does not validate, the document is rejected. To return to the phone book example, if your application validated its input files against a phone book DTD, you would have been spared the problems of debugging your program and correcting your friend’s XML, because your application would have rejected the document as being invalid.

XML Namespaces

XML 1.0 lets developers create their own elements and attributes, but leaves open the potential for overlapping names. For example, a bookstore’s customer file might use `<title>` to hold a value like `Mrs.` or `Dr.`, whereas its catalog document might use `<title>` to hold a value like `Foundation's Edge`. The Namespaces in XML specification (which can be found at `http://www.w3.org/TR/REC-xml-names/`) provides a mechanism developers can use to identify particular vocabularies by using Uniform Resource Identifiers (URIs).

In our hypothetical situtation, the bookstore may need to combine markup from these vocabularies when preparing an invoice. In order to distinguish the two `<title>` elements, each vocabulary is assigned a unique URI. The URI for the customer list might be `http://bookstore.example.com/clients` and the catalog’s URI might be `http://bookstore.example.com/catalogML`. To be complete, the bookstore has a third vocabulary for the elements and attributes specific to the invoice: `http://bookstore.example.com/invoiceML`. Each of these URIs is associated with a *prefix*, which is attached to element name in order to identify which vocabulary it belongs to.

Using namespaces, an invoice might look like this:

```
<inv:invoice xmlns:inv="http://bookstore.example.com/invoiceML"
    xmlns:customer="http://bookstore.example.com/clients"
    xmlns:book="http://bookstore.example.com/catalogML">

    <inv:number>345-0123</inv:number>
    <inv:date>2003-11-08</inv:date>
    <inv:ship-to>
        <customer:title>Dr.</customer:title>
```

[23] See `http://www.docbook.org/`.

```
        <customer:name>V. Thant</customer:name>
        <customer:adresss>2211 Crestview Drive</customer:address>
        <!-- etc -->
    </inv:ship-to>
    <inv:item>
        <book:book isbn="0-345-30898-0">
            <book:title>Foundation's Edge</book:title>
            <book:date>1982</book:date>
            <book:author>Isaac Asimov</book:author>
            <!-- etc. -->
        </book:book>
    </inv:item>
</inv:invoice>
```

In the opening tag, we establish three namespaces. The first one, says that any element or attribute with the prefix `inv` comes from the vocabulary associated with the URI `http://bookstore.example.com/invoiceML`. [24] The prefixes `customer` and `book` are associated with the other two URIs. When we write elements in the invoice document, we write them with the appropriate vocabulary prefix followed by a colon. By using these namespaces and their prefixes, it is now possible for a program to go through invoices and extract only the book titles, avoiding the Doctors and Misters.

Since an OpenDocument file uses information from many different vocabularies, it will establish a large number of these namespace prefixes and URI associations in the opening element. OpenDocument may also use attributes from different vocabularies within an element. When it does so, it puts a namespace prefix on the attribute name as well.

Namespaces are very simple on the surface, but are a well-known field of combat in XML arcana. For more information on namespaces, see Tim Bray's "XML Namespaces by Example," published at `http://www.xml.com/pub/a/1999/01/namespaces.html`. You can also get more information in the books *Learning XML* and *XML in a Nutshell*, published by O'Reilly & Associates.

Tools for Processing XML

Many parsers exist for using XML with many different programming languages. Most of these tools are freely available, the majority being Open Source.

Selecting a Parser

An XML parser typically takes the form of a library of code that you interface with your own program. The program hands the XML over to the parser, and it hands back information about the contents of the XML document. Typically, parsers do this either via *events* or via a *document object model*.

[24] The `<invoice>` element is part of the invoice vocabulary, so it receives the `inv` prefix.

With event-based parsing, the parser calls a function in your program whenever a parse event is encountered. Parse events include things like finding the start of an element, the end of an element, or a comment. Most Java event-based parsers follow a standard API called SAX, which is also implemented for other languages such as Python and Perl. You can find more about SAX at `http://www.megginson.com/SAX/`.

Document object model (DOM) based parsers work in a markedly different way. They consume the entire XML input document and hand back a tree-like data structure that your program can interrogate and alter. The DOM is a W3C standard; documentation is available at `http://www.w3.org/DOM/`.

As XML matures, hybrid techniques that give the best of both worlds are emerging. If you're interested in finding out what's available and what's new for your favorite programming language, then keep an eye on the following online sources:

XML.com Resource Guide: `http://xml.com/pub/resourceguide/`

XMLhack XML Developer News: `http://xmlhack.com/`

Free XML Tools Guide:
`http://www.garshol.priv.no/download/xmltools/`

XSLT Processors

Many XML applications involve transforming one XML document into another or into HTML. The W3C has defined a special language called XSLT for doing transformations. XSLT processors are becoming available for all major programming platforms.

XSLT works by using a *stylesheet*, which contains templates that describe how to transform elements from an XML document. These templates typically specify what XML to output in response to a particular element or attribute. Using a W3C technology called XPath gives you the flexibility not just to say "do this for every 'person' element," but to give instructions as complex as "do this for the third 'person' element whose 'name' attribute is 'Fred' ".

Because of this flexibility, some applications have sprung up for XSLT that aren't really transformation applications at all, but take advantage of the ability to trigger actions on certain element patterns and sequencers. Combined with XSLT's ability to execute custom code via extension functions, the XPath language has enabled applications such as document indexing to be driven by an XSLT processor. You can see a brief introduction to XSLT in Appendix B, The XSLT You Need for OpenDocument.

The W3C specifications for XSLT and XPath can be found at `http://w3.org/TR/xslt` and `http://w3.org/xpath`, respectively.

Appendix B. The XSLT You Need for OpenDocument

The purpose of this appendix is to introduce you to XSLT. A knowledge of XSLT is essential if you wish to easily transform OpenDocument files to text, XHTML, or other XML formats; or to transform XML and XHTML documents to OpenDocument.

The idea behind XSLT is to transform an XML *source document* to an *output document* which may be XML or just plain text. The transformation is accomplished by feeding the source document to an XSLT *transformation stylesheet*.

XPath

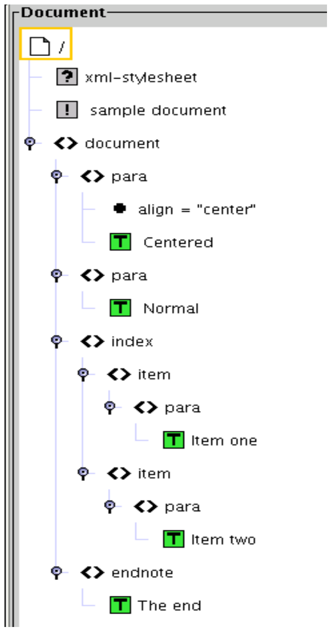
Before we can get into the details of XSLT, we need to talk about how the transformation program views your document and refers to its elements, using a notation called XPath. Take a look at the document in Example B.1, “Sample XML Document” (with line numbers for reference only). XPath (conceptually) represents it as a tree like the one shown in Figure B.1, “Tree Representation of Sample Document”.

Example B.1. Sample XML Document

```
 1  <?xml version="1.0"?>
 2  <?xml-stylesheet type="text/css" href="style.css"?>
 3  <!-- sample document -->
 4  <document>
 5      <para align="center"> Centered </para>
 6      <para> Normal </para>
 7      <index>
 8          <item>
 9              <para>Item one</para>
10          </item>
11          <item>
12              <para>Item two</para>
13          </item>
14      </index>
15      <endnote>The end</endnote>
16  </document>
```

This looks a lot like a file directory listing, and we will begin talking about XPath using this analogy. The highlighted item at the top of the diagram is called the root node of the tree, which corresponds to the root of a UNIX file system. The root node is *not* the same as the root element of the document. As you see, there is a processing instruction node and a comment node that precede the document’s root element. They are part of the document as a whole, and must be represented in the tree.

Figure B.1. Tree Representation of Sample Document



Just as you may select a file from a directory tree by specifying its absolute pathname, you may select any node from the document tree by its absolute path. To select the `<index>` element, the path would be `/document/index`. In a file system, names must be unique within a directory. Thus, a path will select only one file. In an XML document, however, a parent element may have many child elements with the same name. Thus, the absolute XPath expression `/document/para` selects the nodes on lines 5 and 6 of the sample document, and `/document/index/item` selects the elements whose start tags are on lines 8 and 11.

Just as you can create relative path names in a file system (relative to the “current working directory”) XPath allows you to specify nodes by using path names relative to the “context node” in the document tree. If your context node is the `<para>` on line 5, the relative path to the index nodes is simply `index`. If your context node is the `<index>` on line 7, then the path `item/para` will select the nodes on lines 9 and 12. In these cases, as with file systems, every time you take a new step on the path, you look at the children of the context node.

Just as a filesystem path uses `..` (dot dot) to allow you to move up to a parent directory in the directory tree, XPath uses `..` to move to the parent of a node. Thus, if your context node is the `<para>` element on line 12, the relative path name to the `<endnote>` element would be `../../endnote`.

Axes

At this point, we must abandon the filesystem analogy, since XPath gives you many more ways to move around the document tree than just the parent and child relationships. With XPath, you may specify an *axis*, which is the direction in which you want to look as you move to the next step in the path. We have actually been using the child axis as a default; the non-abbreviated syntax for `item/para` is `child::item/child::para`. The following descriptions of XPath axes is adapted from *XML in a Nutshell*, by Elliotte Rusty Harold and W. Scott Means, ISBN 1-56592-580-7.

- `child`—specifies the children of the context node.
- `descendant`—all nodes within the context node. This includes children, grandchildren, great-grandchildren, etc.
- `descendant-or-self`—looks at the same nodes as `descendant`, plus the context node. The abbreviation `//` means "all nodes at the descendant-or-self level."
- `parent`—looks at the parent node of the context node. A node can have only one parent; the root node has no parent. As mentioned before, `..` is the abbreviation for "the node at the parent level."
- `ancestor`—the parent, grandparent, etc. of the context node.
- `ancestor-or-self`—the context node, its parent, grandparent, etc. of the context node.
- `following-sibling`—all nodes that follow the context node and have the same parent as the context node.
- `preceding-sibling`—all nodes that precede the context node and have the same parent as the context node.
- `following`—all nodes that follow the context node in the document, **not** including descendant nodes.
- `preceding`—all nodes that precede the context node in the document, **not** including ancestor nodes.
- `attribute`—the attributes of the context node. The abbreviation for the attribute axis is `@`. Attributes are the illegitimate children of XPath; they know who their parents are, but the parents won't acknowledge them unless you ask for them specifically. In technical terms, attributes are not considered as children of a node and can only be accessed on the attribute axis. However, an attribute node's parent is the element that it belongs to.
- `namespace`—the namespaces that are in scope on the context node.
- `self`—the current node. Its XPath abbreviation is `.` (dot), just as a dot means the current directory in a filesystem.

In an XPath expression, you select nodes by giving the axis and the name of the node(s) you want. You may also use these constructions: *axis*`::*` to mean “all element nodes along an axis,” `@*` to mean “all attribute nodes of the context node,” and *axis*`::node()` to mean “all nodes, including text and comments, attributes, and namespaces, along an *axis*. Table B.1, “Examples of XPath Axes” gives some examples in terms of line numbers in Example B.1, “Sample XML Document”.

Table B.1. Examples of XPath Axes

Current Node	XPath Expression	Node(s) selected
9	`ancestor::*`	8, 7, 4, and root node
4	`descendant::para`	5, 6, 9, 11
12	`*`	The text `item two`
5	`@*`	The attribute `align="center"`
8	`following::para`	12

Predicates

But wait, there’s more! You can also select nodes depending on a condition. This condition is known as a *predicate* (since the selection is “predicated upon” the condition). The predicate is enclosed in square brackets, and may also contain an XPath expression. Thus, to find all `<para>` elements that have an `align` attribute with the value `center`, you would say `//para[@align="center"]`, or, without abbreviations, `/descendant-or-self::para[attribute::align="center"]`. If you were at line 15 of Example B.1, “Sample XML Document” and wanted to find all the preceding `<para>` elements that had `<item>` parents, you would say `preceding::para[parent::item]`. A predicate does not require a relational operation like = or > or <=; if the node exists, the predicate is true. If it doesn’t, the predicate is false.

Note

The expression `preceding::para[parent::item]` selects the `<para>` nodes, *not* the `<item>` node. The predicate looks for the `parent::item`, but does not change the context node.

If the predicate consists of a number *n*, then the predicate matches the *n*th element in the set of selected nodes. The first node is numbered one, not zero.

XPath’s ability to reach any part of a document from any other part of the document allows XSLT to perform powerful and radical transformations on documents.

XSLT

An XSLT stylesheet consists of a series of *templates*, which tell the transformation engine what to do when it encounters certain items. For example, a template might express the idea of “whenever you find a `<para>` element in the source document, put a `<p>` element into the output document.” We’ll see these in action in a while, but first, let’s look at the simplest possible transformation.

XSLT Default Processing

Here is the simplest possible XSLT stylesheet, consisting of just a beginning and ending tag:

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    version="1.0">
</xsl:stylesheet>
```

Although you may not see anything between those tags, XSLT has inserted some default templates for you. Here are the two most important ones:

```
<xsl:template match="*|/">
    <xsl:apply-templates/>
</xsl:template>

<xsl:template match="text()|@*">
    <xsl:value-of select="."/>
</xsl:template>
```

The first template says, “if the current node is an element node (`*`) or (`|`) the root node (`/`), then visit all its children and see if you have any templates that apply to them.” The second template says, “if the current node is a text node (`text()`) or an attribute node (`@*`), output the value of that node (`.`).” Other default templates tell XSLT to ignore processing instructions, comments, and namespace nodes.

If we use this transformation on Example B.1, “Sample XML Document”, XSLT will start at the root node and visit all its children. The first child is a processing instruction, which is ignored. The second child is a comment, which is also ignored. The third child is the `<document>` node. Since we have no template of our own, the first default template (`*|/`) is the only one that matches, so we visit the `<document>`’s children. The first child is a `<para>`, and, since we haven’t provided a template to match it, the first default matches again, and we visit `<para>`’s children. Since attributes are not considered as children, we skip the `align` attribute, and find the text node `Centered`. The second default template matches that text node, so the value `Centered` is written to the output file.

Proceeding in this manner, we see that the empty transformation will create a document consisting of only the text contained therein.

Note

If you run this transformation, you will see that the output contains many blank lines. This is because the newline and tabs between, say, lines 4 and 5, actually create a text node, and those text nodes go into the output document via the default templates. We conveniently ignored these “whitespace-only” text nodes when producing the tree diagram in Figure B.1, “Tree Representation of Sample Document”, but now we have to face their cold reality.

Adding Your Own Templates

Now let’s add some templates to handle things ourselves and create an XHTML file as the output from our sample file.

Example B.2. Simple Templates

```
<xsl:template match="document">
    <html>
        <head>
            <title>My Document</title>
        </head>
        <body>
            <xsl:apply-templates/>
        </body>
    </html>
</xsl:template>

<xsl:template match="para">
    <p>
        <xsl:apply-templates/>
    </p>
</xsl:template>

<xsl:template match="index">
    <h1>Index</h1>
    <ul>
    <xsl:apply-templates/>
    </ul>
</xsl:template>

<xsl:template match="item">
    <li>
        <xsl:apply-templates/>
    </li>
</xsl:template>

<xsl:template match="endnote">
    <h3><xsl:apply-templates/></h3>
</xsl:template>
```

Let’s examine the first template in detail. This says: “Whenever you find a `<document>` element, I want you to emit these items: [25] `<html>`, `<head>`, `<title>`, the text `My Document`, `</title>`, `</head>`, and `<body>`.

“Now go visit all the children of this node and apply any templates you have for them. Once you have finished that, …

“Emit a `</body>` and `</html>` into the output. That is all that needs to be done to completely handle a `<document>` element.”

The `<document>`’s first child is a `<para>`, and it will be handled by the second template that we have added. It says, “Whenever you encounter a `<para>` node, emit a `<p>` into the output, visit all of the children of this node and process them as their templates demand, then emit a `</p>` into the output.” Since the paragraph’s child is a text node, the default template will put the text into the output between the `<p>` and `</p>` tags.

The `<document>`’s next child is an `<index>`, so the appropriate `<xsl:apply-template>` will be applied. It, in turn, uses `<xsl:apply-templates>` to visit all its `<item>` children, which will be handled by the `<xsl:template match="item">`.

After the `<index>` and its descendants are handled, the last child of `<document>` —the `<endnote>` will be handled by the last `<xsl:template>`.

Selecting Nodes to Process

So far, we have been processing all the child nodes indiscriminately with `<xsl:apply-templates/>`. If we wish to visit only specific children, we use a `select` attribute whose value is an XPath expression that selects the children we wish to visit. Thus, if we only wanted the index portion of the document to be converted to XHTML, we would change the first template of Example B.2, “Simple Templates” to read:

```
<xsl:template match="document">
    <html>
        <head>
            <title>My Document</title>
        </head>
        <body>
            <xsl:apply-templates select="index"/>
        </body>
    </html>
</xsl:template>
```

[25] It will also emit newlines and tabs into the output, but let’s ignore that whitespace to make our lives easier.

The important thing to remember about the `select` expression is that the context node is the one that was matched by the `<xsl:template>`. In the preceding example, the context node for the `select="index"` is the `<document>` element currently being processed.

To show a more complicated example of a selection, we need a more complex XML file. Example B.3, "Gradebook Data in XML" shows a student gradebook file represented in XML:

Example B.3. Gradebook Data in XML

```
<?xml version="1.0" encoding="UTF-8"?>
<gradebook>
    <task-list>
        <task date="2003-09-09" id="P01" max="100" recorded="yes"
            type="lab">Program 1</task>
        <task date="2003-09-16" id="Q01" max="100" recorded="yes"
            type="quiz" weight="">Quiz One</task>
    </task-list>
    <student id="S50107">
        <last>Acosta</last> <first>Enrique</first>
        <email>enrique_a@example.com</email>
        <result ref="P01" score="100"/>
        <result ref="Q01" score="80"/>
    </student>
    <student id="S63451">
        <last>Barnett</last> <first>Teresa</first>
        <email>tbarnett@example.com</email>
        <result ref="P01" score="85"/>
        <result ref="Q01" score="92"/>
    </student>
    <student id="S10907">
        <last>Nguyen</last> <first>Thai</first>
        <email/>
        <result ref="P01" score="95"/>
        <result ref="Q01" score="75"/>
    </student>
</gradebook>
```

Our goal is to list the results for all the students who have active email addresses and have taken Quiz 1. The following `select` will choose the appropriate `<result>` elements:

```
gradebook/student[email!='']/result[@ref='Q01']
```

When you get a complex XSLT expression like this, it is probably best read from right to left. It will select all:

- `<result>` elements whose `@ref` attribute equals `Q01`,
- which belong to a `<student>` whose child `<email>` element is not null,
- which is a child of the `<gradebook>` element.

Example B.4, "Complex XPath Selector" shows the XSLT stylesheet.

Example B.4. Complex XPath Selector

```
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:output method="text"/> ❶

<xsl:template match="/">
    <xsl:text>Results for Quiz 1 ❷
</xsl:text>

    <xsl:apply-templates
        select="gradebook/student[email!='']/result[@ref='Q01']"/>
</xsl:template>

<xsl:template match="result">
    <xsl:value-of select="../first"/> ❸
    <xsl:text> </xsl:text>
    <xsl:value-of select="../last"/>
    <xsl:text> </xsl:text>
    <xsl:value-of select="@score"/><xsl:text>
    </xsl:text>
</xsl:template>

</xsl:stylesheet>
```

❶ We don't want an XML file as output, we just want plain text.

❷ The content of `xsl:text` is output verbatim into the output file—including the newline before the closing tag. If you are producing a plain text file, `<xsl:text>` gives you control over whitespace. When you put text into a template that is not surrounded by `<xsl:text>`, then its whitespace will be determined by the underlying XML/XSLT engine's whitespace processing.

❸ `<xsl:value-of>` outputs the text content of the selected node. If the select refers to more than one node, it outputs the text of the first node only. In this case, the XPath selection reaches up to the parent `<student>` node via `..` and then selects its child `<first>` element.

Conditional Processing in XSLT

Although the ability to use predicates gives us some flexibility with transformations, there are times we would like to do totally different actions depending upon some condition. That is why XSLT provides the `<xsl:if>` and `<xsl:choose>` elements. We will now modify Example B.4, "Complex XPath Selector" to choose all students, regardless of their email status, and print a message next to the scores of those students who have no email. The relevant changes are in Example B.5, "Using <xsl:if>".

Example B.5. Using <xsl:if>

```
<xsl:template match="/">
    <xsl:text>Results for Quiz 1
</xsl:text>
```

```
    <xsl:apply-templates
        select="gradebook/student/result[@ref='Q01']"/>
</xsl:template>

<xsl:template match="result">
    <xsl:value-of select="../first"/>
    <xsl:text> </xsl:text>
    <xsl:value-of select="../last"/>
    <xsl:text> </xsl:text>
    <xsl:value-of select="@score"/>
    <xsl:if test="../email = ''">
        <xsl:text> ** has no email **</xsl:text>
    </xsl:if>
    <xsl:text>
</xsl:text>
</xsl:template>
```

(We could have also used `preceding-sibling::email` as the value of the `test` attribute.)

If you are a programmer, your next instinct is to ask "where is the `<xsl:else>` element? Sorry, but there is none. If you need to do a multi-way test, you need to use `<xsl:choose>`, which contains one or more `<xsl:when>` elements, each of which gives a condition. `<xsl:otherwise>` is the branch taken if none of the other conditions matches. Example B.6, "Example of <xsl:choose>" shows the relevant portion of an XSLT stylesheet that assigns an evaluation to the quiz scores. Because this is XML, we must encode a less than sign as `&lt;`. For symmetry, we encode the greater than sign as `&gt;`.

Example B.6. Example of <xsl:choose>

```
<xsl:template match="result">
    <xsl:value-of select="../first"/>
    <xsl:text> </xsl:text>
    <xsl:value-of select="../last"/>
    <xsl:text> </xsl:text>
    <xsl:value-of select="@score"/>
    <xsl:choose>
        <xsl:when test="@score &gt;= 90">
            <xsl:text> Excellent!</xsl:text>
        </xsl:when>
        <xsl:when test="@score &gt;= 80">
            <xsl:text> Good.</xsl:text>
        </xsl:when>
        <xsl:when test="@score &gt;= 70">
            <xsl:text> Fair.</xsl:text>
        </xsl:when>
        <xsl:otherwise>
            <xsl:text> Feh.</xsl:text>
        </xsl:otherwise>
    </xsl:choose>
    <xsl:text>
</xsl:text>
</xsl:template>
```

XSLT Functions

It is possible to do arithmetic calculations in XSLT. For example, if you wanted to output the average of the quiz and program scores in the gradebook example, you could use this expression:

```
<xsl:template match="student">
    <xsl:value-of select="(result[1]/@score + result[2]/@score) div 2"/>
</xsl:template>
```

Notice that we need to use `div` for division, since the forward slash is already used to separate steps in a path. If you are doing subtraction, make sure you surround the minus sign with whitespace, because a hyphen could legitimately be part of an element name.

In addition to the normal arithmetic operators, XSLT is provided with a panoply of mathematical and string functions. (You have no idea of how long I have wanted to use "panoply" in a document.) Here are some of the more important ones.

`concat()`
: Takes a variable number of arguments, and returns the result of concatenating them. In the preceding template, we could have done the output of the student's name from the gradebook with this:

```
<xsl:value-of select="concat(../first ,' ', ../last)"/>
```

`count()`
: Takes an XPath expression as its argument and returns the number of items in that node set. Thus, to print the number of tasks in the gradebook, we could say:

```
<xsl:value-of select="count(/gradebook/task-list/task)"/>
```

`last()`
: Returns the number of the last node in the set being matched by the current template. Note that `count()` requires an argument; `last()` does not.

`name()`
: Returns the fully qualified name of the current node. If the current node doesn't have a name, this returns the null string. This includes any namespace prefix. If you don't want the prefix, use the `local-name()` function instead.

`normalize-space()`
: Strips leading and trailing whitespace from its argument, and replaces any run of internal whitespace with a single space.

`position()`
: Tells which node in the nodeset is currently being processed. This example will print a list of student names followed by the phrase *n* `of` *`total`*

```
      <xsl:template match="student">
          <xsl:value-of select="concat(first, ' ', last)"/>
          <xsl:text> student </xsl:text>
          <xsl:value-of select="position()"/>
          <xsl:text> of </xsl:text>
          <xsl:value-of select="last()"/><xsl:text>
      </xsl:text>
      </xsl:template>
```

`substring()`

Takes two or three arguments. The first argument is a string to be "sliced." The second argument is the starting character of the substring. The first character in a string is character number one, not zero! The third argument is the number of characters in the substring. If no third argument is given, then the substring extends to the end of the original string.

`sum()`

Converts the contents of the current node set to numbers and returns the sum of the contents of the nodes. If any of these is non-numeric, then the result will be `NaN` (Not a Number).

XSLT Variables

Well, if you can perform arithmetic operations and use functions, you must have variables. Indeed, XSLT has variables, but they aren't variables in the traditional C/C++/Java sense.

You may declare a variable with the `<xsl:variable>` element. The variable name is given in the `name` element. You set the variable with either a `select` attribute or the content of the element. When you use a variable, you precede its name with a dollar sign.

A variable can contain more than just a simple string or number; it can contain a whole set of nodes, as shown in Example B.7, "Setting Variables".

Example B.7. Setting Variables

```
<xsl:template match="gradebook"
<!-- set a variable named courseName to "Intro to XML" -->
<xsl:variable name="courseName">Intro to XML</xsl:variable>

<!-- set a variable named quizScores to the set of all the
     quiz score attribute nodes -->
<xsl:variable name="quizScores"
    select="student/result[@ref='Q01']/@score"/>

<!-- produce output -->
<xsl:value-of select="$courseName"/> <xsl:text> </xsl:text>
<xsl:value-of select="sum($quizScores) div count($quizScores)"/>
    <xsl:apply-templates select="gradebook/student"/>

</xsl:template>
```

The scope of the variables in the preceding example is the enclosing `<xsl:template>` element. A variable declared at the top level of the XSLT stylesheet is global and available to all the templates.

XSLT variables are “variable” only in the sense that their values may change each time the template that uses them is invoked. For the duration of the template, the value that is first established cannot be changed.

Named Templates, Calls, and Parameters

Sometimes you may need to emit the same output at several different stages of a transformation. Rather than duplicate the markup, you may create a named template that contains that markup and then call upon it, much as you would use a traditional subroutine. Example B.8, “Named XSLT Template” shows a template that inserts “boilerplate” markup into an output XHTML document. The starting `<xsl:template>` element has a `name` attribute rather than a `match` attribute.

Example B.8. Named XSLT Template

```
<xsl:template name="back-to-top">
    <hr width="75%" />
    <a href="#top">Return to top of page</a>
    <hr width="75%" />
</xsl:template>
```

To invoke this template from anywhere else in the transformation, you need but say `<xsl:call-template name="back-to-top"/>`.

Just as subroutines have arguments, templates can have parameters that modify the way they work. Parameters to a template are named with the `<xsl:param>` element. The content of this element is the default value for the parameter, in case the call does not pass one. Consider Example B.9, “XLST Template with Parameters”, a template which displays a money amount in red if it is negative, black if zero or greater. If no parameter is passed to this template, it will presume a value of zero.

Example B.9. XLST Template with Parameters

```
<xsl:template name="show-money">
    <xsl:param name="amount">0</xsl:param>
    <xsl:choose>
        <xsl:when test="$amount &lt; 0">
            <span style="color:red">
                <xsl:text>$</xsl:text>
                <xsl:value-of select="-$amount"/>
            </span>
        </xsl:when>
        <xsl:otherwise>
            <span style="color:black">
                <xsl:text>$</xsl:text>
                <xsl:value-of select="$amount"/>
            </span>
        </xsl:otherwise>
    </xsl:choose>
</xsl:template>
```

The parameter is passed using the `<xsl:with-param>` element. The parameter value is either the content of the element or the result of its `select` attribute. Example B.10, “Calling an XSLT Template with a Parameter” shows three calls to this template; two with absolute amounts and one from a selected element’s content.

Example B.10. Calling an XSLT Template with a Parameter

```
<xsl:template match="/account">
    <xsl:call-template name="show-money">
        <xsl:with-param name="amount">12.34</xsl:with-param>
    </xsl:call-template>

    <xsl:text> </xsl:text>

    <xsl:call-template name="show-money">
        <xsl:with-param name="amount">-56.78</xsl:with-param>
    </xsl:call-template>

    <xsl:text> </xsl:text>

    <xsl:call-template name="show-money">
        <xsl:with-param name="amount" select="receivable/@qty" />
    </xsl:call-template>
</xsl:template>
```

This brief summary has only touched the surface of XSLT. *XSLT* by Doug Tidwell, ISBN 0-596-00053-7, is an excellent resource for learning more.

Appendix C. Utilities for Processing OpenDocument Files

As we were writing this book, we developed some utilities to make it easier to manipulate OpenDocument files. We hope they are equally useful to you.

An XSLT Transformation

OpenDocument uses the JAR format. Rather than having to unjar each document before running an XSLT transformation on it, we wrote this program, which lets you perform a transformation on a member of a JAR file without having to expand it. It also lets you create a JAR file (without a manifest) as output, if your output is intended to be used as an OpenDocument file.

Getting Rid of the DTD

One other thing prevents us from using a standard transform program such as those provided with Xalan to process the `manifest.xml` file in an OpenDocument file. The manifest contains a `<!DOCTYPE>` declaration that references a file named `Manifest.dtd`. DTD files are used in validating documents. Even though we aren't doing validation on the files, Xalan tries to resolve the file reference and fails, as `Manifest.dtd` does not exist in the JAR file before or after expansion. Example C.1, "Entity Resolver to Ignore DTDs" is a custom `EntityResolver` which eliminates this problem by providing an empty DTD whenever Xalan tries to resolve an external entity whose name ends with `.dtd`.

[This is file `ResolveDTD.java` in directory `appc` in the downloadable example files.]

Example C.1. Entity Resolver to Ignore DTDs

```
import org.xml.sax.EntityResolver;
import org.xml.sax.InputSource;
import java.io.StringReader;
public class ResolveDTD implements EntityResolver {
   public InputSource resolveEntity (String publicId, String systemId)
   {
        if (systemId.endsWith(".dtd"))
        {
            StringReader stringInput =
                new StringReader(" ");
            return new InputSource(stringInput);
        }
        else
        {
            return null;    // default behavior
        }
   }
}
```

The Transformation Program

Now that we have overcome the problem of the phantom DTD, we can write the main transformation program, `ODTransform.java`. It takes the following command line arguments:

`-in`
: This is followed by the name of the input file. If you're operating on a compressed document, then this is the member name of the OpenDocument file, which you must name with the `-inOD` argument.

`-inOD`
: This is followed by the name of a compressed OpenDocument document.

`-xsl`
: This is followed by the name of the XSLT file that does the transformation.

`-out`
: This is followed by the name of the output file. If it is to be a member of an output OpenDocument file, then you must specify that file name with the `-outOD` argument.

`-outOD`
: This is followed by the name of a compressed OpenDocument file that will be produced as output.

`-param`
: This is followed by the name of a parameter to be passed to the transformation, and the value of that parameter.

Thus, if you are transforming a plain file to another plain file, you might have a command line like this:

```
ODTransform -in content.xml -xsl transform.xslt -out result.txt
```

To transform the `content.xml` file inside a document named `myfile.odt`, producing a non-compressed output file, you might have a command line like this:

```
ODTransform -inOD myfile.odt -in content.xml►
   -xsl transform.xsl -out result.txt
```

And, to transform `content.xml` inside a document named `myfile.odt` to produce a new `content.xml` inside a result document named `newfile.odt`, your command line would be:

```
ODTransform -inOD myfile.odt -in content.xml►
   -xsl transform.xslt -outOD newfile.odt -out content.xml
```

When creating an OpenDocument file as output, the program must also create a `META-INF/manifest.xml` file. The extension given in the

```
-outOD
```

parameter will determine the `media-type` in the manifest.

And now, Example C.2, "XSLT Transformation for OpenDocument files", which shows the code, which you will find in file `ODTransform.java` in directory `appc` in the downloadable example files.

Example C.2. XSLT Transformation for OpenDocument files

```
/*
 * ODTransform.java
 * (c) 2003-2005 J. David Eisenberg
 * Licensed under LGPL
 *
 * Program purpose: to perform an XSLT transformation
 * on a member of an OpenDocument file, either
 * after unzipping or while still in its zipped state.
 * Output may go to a normal file or a zipped file.
 */

import javax.xml.transform.TransformerFactory;
import javax.xml.transform.Transformer;
import javax.xml.transform.sax.SAXSource;
import javax.xml.transform.sax.SAXResult;
import javax.xml.transform.stream.StreamSource;
import javax.xml.transform.stream.StreamResult;
import javax.xml.transform.TransformerException;
import javax.xml.transform.TransformerConfigurationException;

import org.xml.sax.XMLReader;
import org.xml.sax.InputSource;
import org.xml.sax.ContentHandler;
import org.xml.sax.ext.LexicalHandler;
import org.xml.sax.SAXException;
import org.xml.sax.helpers.XMLReaderFactory;

import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;

import java.util.Hashtable;
import java.util.jar.JarInputStream;
import java.util.jar.JarOutputStream;
import java.util.jar.JarEntry;
import java.util.Vector;
import java.util.zip.ZipException;

public class ODTransform
{
    String  inputFileName = null;   // input file name, or member
                                    // name...
    String  inputODName = null;     // ...if given an OpenDocument
                                    // input file
    String  outputFileName = null;  // output file name, or member
                                    // name...
    String  outputODName = null;    // ...if given an OpenDocument
                                    // output file
    String  xsltFileName = null;    // XSLT file is always a regular
                                    // file

    Vector  params = new Vector();  // parameters to be passed to
                                    // transform

    public void doTransform( )
    throws TransformerException, TransformerConfigurationException,
         SAXException, ZipException, IOException
    {
       /* Set up the XSLT transformation based on the XSLT file */
       File xsltFile = new File( xsltFileName );
       StreamSource streamSource = new StreamSource( xsltFile );
       TransformerFactory tFactory = TransformerFactory.newInstance();
       Transformer transformer = tFactory.newTransformer
           ( streamSource );

        /* Set up parameters for transform */
        for (int i=0; i < params.size(); i += 2)
        {
            transformer.setParameter((String) params.elementAt(i),
                (String) params.elementAt(i + 1));
        }

        /* Create an XML reader which will ignore any DTDs */
        XMLReader reader = XMLReaderFactory.createXMLReader();
        reader.setEntityResolver( new ResolveDTD() );

        InputSource inputSource;

        if (inputODName == null)
        {
            /* This is an unpacked file. */
            inputSource =
                new InputSource( new FileInputStream( inputFileName
) );
        }
        else
        {
            /* The input file should be a member of an OD file.
               Check to see if the input file name really exists
               within the JAR file */
            JarInputStream jarStream =
                new JarInputStream( new
                    FileInputStream( inputODName ), false );
            JarEntry jarEntry;
            while ( (jarEntry = jarStream.getNextJarEntry() )
                != null &&
                !(inputFileName.equals(jarEntry.getName()) ) )
                // do nothing
                ;
            inputSource = new InputSource( jarStream );
        }

        SAXSource saxSource = new SAXSource( reader, inputSource );
        saxSource.setSystemId( inputFileName );
        if (outputODName == null)
        {
            /* We want a regular file as output */
            FileOutputStream outputStream =
                new FileOutputStream( outputFileName );
            transformer.transform( saxSource,
                new StreamResult( outputStream ) );
            outputStream.close();
        }
        else
        {
            /* The output file name is the name of a member of
               a JAR file (which we will build without a manifest) */
            JarOutputStream jarStream =
                new JarOutputStream( new FileOutputStream
                ( outputODName ) );
            JarEntry jarEntry = new JarEntry( outputFileName );
            jarStream.putNextEntry( jarEntry );
            transformer.transform( saxSource,
                new StreamResult( jarStream ) );

            /* Close the member file and the JAR file
               to complete the file */
            jarStream.closeEntry();

            createManifestFile( jarStream );

            /* Close the JAR file to complete the file */
            jarStream.close();
        }
    }

    /* Check to see if the command line arguments make sense */
    private void checkArgs( String[] args )
    {
        int     i;

        if (args.length == 0)
        {
            showUsage( );
            System.exit( 1 );
        }
        i = 0;
        while  ( i < args.length )
        {
            if (args[i].equalsIgnoreCase("-in"))
            {
                if  ( i+1 >= args.length)
                {
                    badParam("-in");
                }
                inputFileName = args[i+1];
                i += 2;
            }
            else if (args[i].equalsIgnoreCase("-out"))
            {
                if  ( i+1 >= args.length)
                {
                    badParam("-out");
                }
                outputFileName = args[i+1];
                i += 2;
            }
            else if (args[i].equalsIgnoreCase("-xsl"))
            {
                if ( i+1 >= args.length)
                {
                    badParam("-xsl");
                }
                xsltFileName = args[i+1];
                i += 2;
            }
            else if (args[i].equalsIgnoreCase("-inod"))
            {
                if ( i+1 >= args.length)
                {
                    badParam("-inOD");
                }
                inputODName = args[i+1];
                i += 2;
            }
            else if (args[i].equalsIgnoreCase("-outod"))
            {
                if ( i+1 >= args.length)
                {
                    badParam("-outOD");
                }
                outputODName = args[i+1];
                i += 2;
            }
            else if (args[i].equalsIgnoreCase("-param"))
            {
                if ( i+2 >= args.length)
                {
                    badParam("-param");
                }
                params.addElement( args[i+1] );
                params.addElement( args[i+2] );
                i += 3;
            }
            else
            {
                System.out.println( "Unknown argument " + args[i] );
                System.exit( 1 );
            }
        }

        if (inputFileName == null)
        {
            System.out.println("No input file name specified.");
            System.exit( 1 );
        }
        if (outputFileName == null)
        {
            System.out.println("No output file name specified.");
            System.exit( 1 );
        }
        if (xsltFileName == null)
        {
            System.out.println("No XSLT file name specified.");
            System.exit( 1 );
        }
    }

    /* If not enough arguments for a parameter, show error and exit */
    private void badParam( String paramName )
    {
        System.out.println("Not enough parameters to " + paramName);
        System.exit(1);
    }

    /*
        Creates the manifest file for a compressed OpenDocument
        file.  The mType array contains pairs of filename
        extensions and corresponding mimetypes.  The comparison
        to find the extension is done in a case-insensitive manner.
    */
    private void createManifestFile( JarOutputStream jarStream )
    {
        String [] mType = {
        "odt", "application/vnd.oasis.opendocument.text",
        "ott", "application/vnd.oasis.opendocument.text-template",
        "odg", "application/vnd.oasis.opendocument.graphics",
        "otg",
           "application/vnd.oasis.opendocument.graphics-template",
        "odp", "application/vnd.oasis.opendocument.presentation",
        "otp",
           "application/vnd.oasis.opendocument.presentation-template",
        "ods", "application/vnd.oasis.opendocument.spreadsheet",
        "ots",
            "application/vnd.oasis.opendocument.spreadsheet-template",
        "odc", "application/vnd.oasis.opendocument.chart",
        "otc", "applicationvnd.oasis.opendocument.chart-template",
        "odi", "application/vnd.oasis.opendocument.image",
        "oti", "applicationvnd.oasis.opendocument.image-template",
        "odf", "application/vnd.oasis.opendocument.formula",
        "otf", "applicationvnd.oasis.opendocument.formula-template",
        "odm", "application/vnd.oasis.opendocument.text-master",
        "oth", "application/vnd.oasis.opendocument.text-web",
        };

        JarEntry jarEntry;

        int dotPos;
        String extension;
        String mimeType = null;
        String outputStr;

        dotPos = outputODName.lastIndexOf(".");
        extension = outputODName.substring( dotPos + 1 );
        for (int i=0; i < mType.length && mimeType == null; i+=2)
        {
            if (extension.equalsIgnoreCase( mType[i] ))
            {
                mimeType = mType[i+1];
            }
        }

        if (mimeType == null)
        {
            System.err.println("Cannot find mime type for extension "
                + extension );
            mimeType = "UNKNOWN";
        }

        try
        {
            jarEntry = new JarEntry( "META-INF/manifest.xml");
            jarStream.write( "<?xml version=\"1.0\" encoding=
              \"UTF-8\"?>"
                .getBytes() );
            jarStream.write( "<!DOCTYPE manifest:manifest PUBLIC
              \"-//OpenOffice.org//DTD Manifest 1.0//EN\"
              \"Manifest.dtd\">"
                .getBytes() );
            jarStream.write("<manifest:manifest xmlns:manifest=
              \"urn:oasis:names:tc:opendocument:xmlns:manifest:1.0\">"
                .getBytes() );

            outputStr = "<manifest:file-entry manifest:media-type=\""
+
                mimeType + "\" manifest:full-path=\"/\"/>";
            jarStream.write( outputStr.getBytes() );

            outputStr = "<manifest:file-entry manifest:media-type=
               \"text/xml\" manifest:full-path=\"" + outputFileName
               + "\"/>";
            jarStream.write( outputStr.getBytes() );
            jarStream.write("</manifest:manifest>".getBytes() );
            jarStream.closeEntry();
        }
        catch (IOException e)
        {
            System.err.println("Cannot write file:");
            System.err.println( e.getMessage() );
        }
    }

    /* If no arguments are provided, show this brief help section */
    private void showUsage( )
    {
        System.out.println("Usage: ODTransform options");
        System.out.println("Options:");
        System.out.println("   -in inputFilename");
        System.out.println("   -xsl transformFilename");
        System.out.println("   -out outputFilename");
        System.out.println("If the input filename is within an
            OpenDocument file, then:");
        System.out.println("   -inOD inputOpenDocFileName");
        System.out.println("If you wish to output an OpenDocument
            file, then:");
        System.out.println("   -outOD outputOpenDocumentFileName");
        System.out.println( );
        System.out.println("Argument names are case-insensitive.");
    }

    public static void main(String[] args)
    {
        ODTransform transformApp = new ODTransform( );
        transformApp.checkArgs( args );
        try {
            transformApp.doTransform( );
        }
        catch (Exception e)
        {
            System.out.println("Unable to transform");
            System.out.println(e.getMessage());
        }
    }
}
```

Transformation Script

You need to set the class path correctly in order to use this program; you may either set your CLASSSPATH system environment variable, or set it up in a shell script. Example C.3, "Invoking the Transform Program" shows the bash shell script that I used. You will find it in file `odtransform.sh` in directory `appc` in the downloadable example files.

Example C.3. Invoking the Transform Program

```
java -cp /usr/local/xmljar/xalan-j_2_6_0/bin/xalan.jar:\
/usr/local/xmljar/xalan-j_2_6_0/bin/xercesImpl.jar:\
/usr/local/xmljar/xalan-j_2_6_0/bin/xml-apis.jar:\
/usr/local/xmljar/xalan-exts/utils.jar:\
/usr/local/odbook/ODTransform \
ODTransform $1 $2 $3 $4 $5 $6 $7 $8 $9 ${10} ${11} ${12} ${13} ${14} \
${15} ${16} ${17}
```

Using XSLT to Indent OpenDocument Files

As an application of the preceding script, we present an alternate method of indenting the unpacked files via a simple XSLT transformation. Example C.4, "XSLT Transformation for Indenting" shows this transformation, which simply copies the entire document tree while setting `indent` to `yes` in the `<xsl:output>` element.

Example C.4. XSLT Transformation for Indenting

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:output method="xml" indent="yes"/>

<xsl:template match="/">
    <xsl:copy-of select="."/>
</xsl:template>

</xsl:stylesheet>
```

We now present a Perl program to invoke this transformation on all the XML files in an unpacked OpenDocument file. We will need to set two paths: one to the transformation script, and one to the location of the preceding XSLT transformation. Make sure you use absolute paths for setting variables `$script_location` and `$transform_location`, because `find()` changes directories as it traverses the directory tree. This is file `od_indent.pl` in the `appc` directory in the downloadable example files.

Example C.5. Program to Indent OpenDocument Files via XSLT

```
#!/usr/bin/perl

use File::Find;

#
#   This program indents XML files within a directory.
#   a simple XSLT transform is used to indent the XML.
#

#
#   Path where you have installed the OpenDocument transform script.
#
$script_location = "/your/path/to/odtransform.sh";

#
#   Path where you have installed the XSLT transformation.
#
$transform_location = "/your/path/to/od_indent.xsl";

if (scalar @ARGV != 1)
{
    print "Usage: $0 directory\n";
    exit;
}

if (!-e $script_location)
{
    print "Cannot find the transform script at $script_location\n";
    exit;
}

if (!-e $transform_location)
{
    print "Cannot find the XSLT transformation file at " ,
        "$transform_location\n";
    exit;
}

$dir_name = $ARGV[0];

if (!-d $dir_name)
{
    print "The argument to $0 must be the name of a directory\n";
    print "containing XML files to be indented.\n";
    exit;
}

#
```

```
#   Indent all XML files.
#
find(\&indent, $dir_name);

#   Warning:
#   This subroutine creates a temporary file with the format
#   __tempnnnn.xml, where nnnn is the current time( ). This
#   will avoid name conflicts when used with OpenOffice.org documents,
#   even though the technique is not sufficiently robust for general
use.
#
sub indent
{
    my $xmlfile = $_;
    my $command;
    my $result;
    if ($xmlfile =~ m/\.xml$/)
    {
        $time = time();
        print "Indenting $xmlfile\n";
        $command = "$script_location " .
            "-in $xmlfile -xsl $transform_location " .
            "-out __temp$time.xml";
        $result = system( $command );
        if ($result == 0 && -e "__temp$time.xml")
        {
            unlink $xmlfile;
            rename "__temp$time.xml", $xmlfile;
        }
        else
        {
            print "Error occurred while indenting $xmlfile\n";
        }
    }
}
```

This process may insert newlines in text as well as between elements. In cases where elements contain other elements, this is not a problem, as OpenDocument ignores whitespace between elements. When expanding text elements, though, the extra newlines could cause extra spaces to appear when repacking the document. Thus, you should use this method to indent the XML document only when you do *not* want to repack the resulting files.

An XSLT Framework for OpenDocument files

When using XLST with OpenDocument files, you will want to make sure you have declared all the appropriate namespaces. Rather than selecting exactly the namespaces that your document uses, we provide all of the namespaces for OpenDocument in Example C.6, “XSLT Framework for Transforming OpenDocument”, which you may use as a framework for your transformations. This is file `framework.xsl` in directory `appc` in the downloadable example files.

Example C.6. XSLT Framework for Transforming OpenDocument

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns:office="urn:oasis:names:tc:opendocument:xmlns:office:1.0"
    xmlns:meta="urn:oasis:names:tc:opendocument:xmlns:meta:1.0"
    xmlns:config="urn:oasis:names:tc:opendocument:xmlns:config:1.0"
    xmlns:text="urn:oasis:names:tc:opendocument:xmlns:text:1.0"
    xmlns:table="urn:oasis:names:tc:opendocument:xmlns:table:1.0"
    xmlns:draw="urn:oasis:names:tc:opendocument:xmlns:drawing:1.0"
    xmlns:presentation="urn:oasis:names:tc:opendocument:xmlns:►
presentation:1.0"
    xmlns:dr3d="urn:oasis:names:tc:opendocument:xmlns:dr3d:1.0"
    xmlns:chart="urn:oasis:names:tc:opendocument:xmlns:chart:1.0"
    xmlns:form="urn:oasis:names:tc:opendocument:xmlns:form:1.0"
    xmlns:script="urn:oasis:names:tc:opendocument:xmlns:script:1.0"
    xmlns:style="urn:oasis:names:tc:opendocument:xmlns:style:1.0"
    xmlns:number="urn:oasis:names:tc:opendocument:xmlns:datastyle:1.0"
    xmlns:anim="urn:oasis:names:tc:opendocument:xmlns:animation:1.0"

    xmlns:dc="http://purl.org/dc/elements/1.1/"
    xmlns:xlink="http://www.w3.org/1999/xlink"
    xmlns:math="http://www.w3.org/1998/Math/MathML"
    xmlns:xforms="http://www.w3.org/2002/xforms"

    xmlns:fo="urn:oasis:names:tc:opendocument:xmlns:►
xsl-fo-compatible:1.0"
    xmlns:svg="urn:oasis:names:tc:opendocument:xmlns:►
svg-compatible:1.0"
    xmlns:smil="urn:oasis:names:tc:opendocument:xmlns:►
smil-compatible:1.0"

    xmlns:ooo="http://openoffice.org/2004/office"
    xmlns:ooow="http://openoffice.org/2004/writer"
    xmlns:oooc="http://openoffice.org/2004/calc"
>

<xsl:template match="/office:document-content">
    <xsl:apply-templates/>
</xsl:template>

</xsl:stylesheet>
```

OpenDocument White Space Representation

If you are creating an OpenDocument file from a file where white space has been preserved, you will have to convert runs of spaces into `<text:s>` elements, and convert tabs and line feeds into `<text:tab-stop>` and `<text:line-break>` elements. This task is not easily done in native XSLT. Example C.7, "Transforming Whitespace to OpenDocument XML" is a Java extension for Xalan which will do what you need. You will note that we create elements and attributes complete with namespace prefix. This is certainly not a recommended practice, but `createElementNS()` and `setAttributeNS()` create `xmlns` attributes rather than a prefixed name. You will find this Java code in file `ODWhiteSpace.java` in directory `appc` in the downloadable example files.

Example C.7. Transforming Whitespace to OpenDocument XML

```
import org.w3c.dom.Document;
import org.w3c.dom.Element;
import org.w3c.dom.Node;
import org.w3c.dom.NodeList;
import org.w3c.dom.Text;
import org.apache.xpath.NodeSet;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.ParserConfigurationException;

public class ODWhiteSpace {

    public ODWhiteSpace ()
    {}

    public static NodeList compressString( String str )
    {
        ODWhiteSpace whiteSpace = new ODWhiteSpace();
        return whiteSpace.doCompress( str );
    }

    private Document tempDoc;       // necessary for creating elements
    private StringBuffer strBuf;    // where non-whitespace accumulates
    private NodeSet resultSet;      // the value to be returned
    private int pos;                // current position in string
    private int startPos;           // where blanks begin accumulating
    private int nSpaces;            // number of consecutive spaces
    private boolean inSpaces;       // handling spaces?
    private char ch;                // current character in buffer
    private char prevChar;          // previous character in buffer
    private Element element;        // element to be added to node list

    /**
     * Create OpenDocument elements for a string.
     * @param str the string to compress.
     * @return a NodeList for insertion into an OpenDocument file
    */
    public NodeList doCompress( String str )
    {
        if (str.length() == 0)
        {
            return null;
        }
        tempDoc = null;
        strBuf = new StringBuffer( str.length() );


        try
        {
            tempDoc = DocumentBuilderFactory.newInstance().
                newDocumentBuilder().newDocument();
        }
        catch(ParserConfigurationException pce)
        {
            return null;
        }

        resultSet = new NodeSet();
        resultSet.setShouldCacheNodes(true);

        pos = 0;
        startPos = 0;
        nSpaces = 0;
        inSpaces = false;
        ch = '\u0000';

        while (pos < str.length())
        {
            prevChar = ch;
            ch = str.charAt( pos );
            if (ch == ' ')
            {
                if (inSpaces)
                {
                    nSpaces++;
                }
                else
                {
                    emitText( );
                    nSpaces = 1;
                    inSpaces = true;
                    startPos = pos;
                }
            }
            else if (ch == 0x000a || ch == 0x000d)
            {
                if (prevChar != 0x000d) // ignore LF or CR after CR.
                {
                   emitPending( );
                   element = tempDoc.createElement("text:line-break");
                   resultSet.addNode(element);
                }
            }
            else if (ch == 0x09)
            {
                emitPending( );
                element = tempDoc.createElement("text:tab-stop");
                resultSet.addNode(element);
            }
            else
            {
                if (inSpaces)
                {
                    emitSpaces( );
                }
                strBuf.append( ch );
            }
            pos++;
        }

        emitPending( );      // empty out anything that's accumulated
        return resultSet;
    }

    /**
     * Emit accumulated spaces or text
     */
    private void emitPending( )
    {
        if (inSpaces)
        {
            emitSpaces( );
        }
        else
        {
            emitText( );
        }
    }

    /**
     * Emit accumulated text.
     * Creates a text node with currently accumulated text.
     * Side effect: empties accumulated text buffer
     */
    private void emitText( )
    {
      if (strBuf.length() != 0)
      {
         Text textNode = tempDoc.createTextNode( strBuf.toString( ) );
            resultSet.addNode( textNode );
            strBuf = new StringBuffer( );
      }
    }

    /**
     * Emit accumulated spaces.
     * If these are leading blanks, emit only a
     * &lt;text:s&gt; element; otherwise a blank plus
     * a &lt;text:s&gt; element (if necessary)
     * Side effect: sets accumulated number of spaces to zero.
     * Side effect: sets "inSpaces" flag to false
     */
    private void emitSpaces( )
    {
        Integer n;

        if (nSpaces != 0)
        {
            if (startPos != 0)
            {
                Text textNode = tempDoc.createTextNode( " " );
                resultSet.addNode( textNode );
                nSpaces--;
            }

            n = new Integer(nSpaces);
            if (nSpaces >= 1 || startPos == 0)
            {
                element = tempDoc.createElement( "text:s" );
                element.setAttribute( "text:c",
                    (new Integer(nSpaces)).toString( ) );
                resultSet.addNode( element );
            }

            inSpaces = false;
            nSpaces = 0;
        }
    }
}
```

Showing Meta-information Using SAX

This is the same program as Example 2.3, "Program show_meta.pl", except that it uses the XML::SAX module instead of XML::Simple. XML::SAX is a perl module for the Simple API for XML, which interfaces to an event-driven parser. The parser issues many kinds of events as it parses a document; the ones we are interested in are the events that occur when an element starts, when it ends, and when we encounter the element's text content. To use XML::SAX, you must specify a *handler* object, which is a Perl package that contains subroutines that are called when the parser detects events. The handler subroutines receive two parameters: a reference to the parser, and data hash with information about the event. Here are the subroutines that we will implement, the keys from the data hash that we are interested in, and how we will use their values.

`start_element`
: This subroutine is called whenever the parser detects an opening tag for an element. The relevant keys are

  `Name`
  : The name of the element (with namespace prefix)

  `Attributes`
  : The value of this key is yet another hash, whose keys are the attribute names, preceded by their namespace URIs. This value for each of these keys is yet another hash, with keys `Name` and `Value`, whose values are the attribute name and value.

  The program will store the element name in a scalar `$element` and the attributes in a global array `@attributes`. It sets a global scalar `$text` to the null string; this variable will be used to collect all the element's text content.

`characters`

This subroutine is called whenever the parser detects a series of characters within an element. The relevant key is

`Data`
The characters that have been parsed.

The text is concatenated to the end of the `$text` variable. This is necessary because a single sequence of text may generate multiple calls to the character handler.

`end_element`

This subroutine is called whenever the parser detects an opening tag for an element. The relevant key is

`Name`
The name of the element (with namespace prefix).

Upon encountering the end of an element, the program will add the element name as a key in a hash named `%info`. The hash value will be an anonymous array consisting of the `$text` content followed by the `@attributes` array.

Here is the rewritten program, which you will find in file `sax_show_meta.pl` in the `appc` directory in the downloadable example files.

Example C.8. Program sax_show_meta.pl

```
#!/usr/bin/perl

#
#   Show meta-information in an OpenDocument file.
#
use XML::SAX;
use IO::File;
use Text::Wrap;
use Carp;
use strict 'vars';

my $suffix;     # file suffix

my $parser;     # instance of XML::SAX parser
my $handler;    # module that handles elements, etc.
my $filehandle; # file handle for piped input

my $info;       # the hash returned from the parser
my @attributes; # attributes from a returned element
my %attr_hash;  # hash of attribute names and values
#
#   Check for one argument: the name of the OpenDocument file
#
if (scalar @ARGV != 1)
{
    croak("Usage: $0 document");
}
```

```
#
#   Get file suffix for later reference
#
($suffix) = $ARGV[0] =~ m/\.(\w\w\w)$/;

#
#   Create an object containing handlers for relevant events.
#
$handler = MetaElementHandler->new();


#
#   Create a parser and tell it where to find the handlers.
#
$parser =
    XML::SAX::ParserFactory->parser( Handler => $handler);

#
#   Input to the parser comes from the output of member_read.pl
#
$ARGV[0] =~ s/[;|'"]//g;  #eliminate dangerous shell metacharacters
$filehandle = IO::File->new(
   "perl member_read.pl $ARGV[0] meta.xml |" ); ❶

#
#   Parse and collect information.
#
$parser->parse_file( $filehandle );

#
#   Retrieve the information collected by the parser
#
$info = $handler->get_info();   ❷

#
#   Output phase
#
print "Title:       $info->{'dc:title'}[0]\n"
    if ($info->{'dc:title'}[0]);
print "Subject:     $info->{'dc:subject'}[0]\n"
    if ($info->{'dc:subject'}[0]);

if ($info->{'dc:description'}[0])
{
    print "Description:\n";
    $Text::Wrap::columns = 60;
    print wrap("\t", "\t", $info->{'dc:description'}[0]), "\n";
}

print "Created:     ";
print format_date($info->{'meta:creation-date'}[0]);
print " by $info->{'meta:initial-creator'}[0]"
    if ($info->{'meta:initial-creator'}[0]);
print "\n";

print "Last edit:   ";
print format_date($info->{"dc:date"}[0]);
print " by $info->{'dc:creator'}[0]"
    if ($info->{'dc:creator'}[0]);
```

```
print "\n";

#
#   Take attributes from the meta:document-statistic element
#   (if any) and put them into %attr_hash
#
@attributes = @{$info->{'meta:document-statistic'}};

if (scalar(@attributes) > 1)
{
    shift @attributes;
    %attr_hash = @attributes;

    if ($suffix eq "sxw")
    {
        print "Pages:       $attr_hash{'meta:page-count'}\n";
        print "Words:       $attr_hash{'meta:word-count'}\n";
        print "Tables:      $attr_hash{'meta:table-count'}\n";
        print "Images:      $attr_hash{'meta:image-count'}\n";
    }
    elsif ($suffix eq "sxc")
    {
        print "Sheets:      $attr_hash{'meta:table-count'}\n";
        print "Cells:       $attr_hash{'meta:cell-count'}\n"
            if ($attr_hash{'meta:cell-count'});
    }
}

#
#   A convenience subroutine to make dates look
#   prettier than ISO-8601 format.
#
sub format_date
{
    my $date = shift;
    my ($year, $month, $day, $hr, $min, $sec);
    my @monthlist = qw (Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov
Dec);

    ($year, $month, $day, $hr, $min, $sec) =
        $date =~ m/(\d{4})-(\d{2})-(\d{2})T(\d{2}):(\d{2}):(\d{2})/;
    return "$hr:$min on $day $monthlist[$month-1] $year";
}


package MetaElementHandler; ❸

my %element_info;   # the data structure that we are creating
my $element;        # name of element being processed
my @attributes;     # attributes for this element
my $text;           # text content of the element


sub new { ❹
    my $class = shift;
    my %opts = @_;
    bless \%opts, $class;
}

sub reset {
    my $self = shift;
    %$self = ();
}

#
#   Store current element and its attribute.
#
sub start_element
{
    my ($self, $parser_data) = @_;

    my $hashref; ❺
    my $item;       # loop control variable

    $element = $parser_data->{"Name"};

    foreach $item (keys %{$parser_data->{"Attributes"}})
    {
        $hashref =  $parser_data->{"Attributes"}{$item};
        push @attributes, $hashref->{"Name"},  $hashref->{"Value"};
    }

    $text = ""; # no text content yet.
}

#
#   Create an entry into a hash for the element that is ending
#
sub end_element
{
    my ($self, $parser_data) = @_;

    $element = $parser_data->{"Name"};
    $element_info{$element} = [$text, @attributes];
}

#
#   Accumulate element's text content.
#
sub characters
{
    my ($self, $parser_data) = @_;
    $text .= $parser_data->{"Data"}; ❻
}

#   Return a reference to the %info hash
#
sub get_info ❼
{
    my $self = shift;
    return \%element_info;
}
```

❶ XML::SAX doesn't read from file handles opened with the standard Perl `open()` function; you have to use IO::File to create the file handle.

❷ The handler object has accumulated all the information from the `meta.xml` file into a hash. We ask the handler to return a reference to that hash.

❸ XML::SAX wants its handler subroutines to be in a Perl object. The `package` statement serves to “encapsulate” the variables and subroutines. As good citizens, we don’t directly access any of the variables from the main program.

❹ The `new` subroutine completes the work of making this package into a Perl object. The `reset` subroutine is for XML::SAX’s internal use.

❺ The `$hashref` variable is here for convenience; if we didn’t use it, then the `push` statement would be even less readable than it already is.

❻ Note the `.=` operation; since the text inside an element can come from many calls to `characters`, we have to concatentate them all.

❼ This is not an XML::SAX routine; we are providing it so that we can hand a reference to our accumulated data back to the main program.

Creating Multiple Directory Levels

If you need to create multiple directory levels, but your system doesn’t have the equivalent of Linux’s `mkdir --parents` option, use the program shown in Example C.9, “Program to Create Directories”, which is in file `make_directories.pl` in the `appc` directory in the downloadable example files.

Example C.9. Program to Create Directories

```
#
#   Command line parameter: pathname of directory to create
#
#   Creates directory and all intervening levels.
#
use File::Path;

if (scalar @ARGV == 1)
{
    mkpath($ARGV[0], 1, 0755);
}
else
{
    print "Usage: $0 path_to_create\n";
}
```

Appendix D. GNU Free Documentation License

Version 1.2, November 2002

PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document “free” in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of “copyleft”, which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The “Document”, below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as “you”. You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A “Modified Version” of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A “Secondary Section” is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document’s overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The “Invariant Sections” are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The “Cover Texts” are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A “Transparent” copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not “Transparent” is called “Opaque”.

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The “Title Page” means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, “Title Page” means the text near the most prominent appearance of the work’s title, preceding the beginning of the body of the text.

A section “Entitled XYZ” means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as “Acknowledgements”, “Dedications”, “Endorsements”, or “History”.) To “Preserve the Title” of such a section when you modify the Document means that it remains a section “Entitled XYZ” according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document’s license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled “History” in the various original documents, forming one section Entitled “History”; likewise combine any sections Entitled “Acknowledgements”, and any sections Entitled “Dedications”. You must delete all sections Entitled “Endorsements”.

COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an “aggregate” if the copyright resulting from the compilation is not used to limit the legal rights of the compilation’s users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document’s Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled “Acknowledgements”, “Dedications”, or “History”, the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

> Copyright (c) YEAR YOUR NAME. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with...Texts." line with this:

> with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Index